A Survey of Malware Detection Techniques

Nwokedi Idika[†] (nidika@purdue.edu)
Aditya P. Mathur[‡] (apm@purdue.edu)

Department of Computer Science
Purdue University, West Lafayette, IN 47907.

† Research supported by Committee on Institutional Cooperation and General Electric (CIC/GE), Predoctoral Fellowship, and Purdue Doctoral Fellowship (PDF)

‡ Research supported by Arxan Technologies/21STC.R&T Fund

February 2, 2007

Contents

Abstract

Malware is a worldwide epidemic. Studies suggest that the impact of malware is getting worse. Malware detectors are the primary tools in defense against malware. The quality of such a detector is determined by the techniques it uses. It is therefore imperative that we study malware detection techniques and understand their strengths and limitations. This survey examines 45 malware detection techniques and offers an opportunity to compare them against one another aiding in the decision making process involved with developing a secure application/system. The survey also provides a comprehensive bibliography as an aid to researchers in malware detection.

Keywords: anomaly-based, malware, malware detection, malcode, malicious code, malicious software, signature-based.

1 Introduction

Malware has had a tremendous impact on the world as we know it. The rising number of computer security incidents since 1988 [7, 8] suggests that malware is an epidemic. Surfing the World Wide Web with all anti-virus and firewall protection disabled for a day should convince any reader of the widespread and malicious nature of malware. A similar experiment was conducted by the San Diego Supercomputer Center (SDSC) [39]. In December of 1999, SDSC installed Red Hat Linux 5.2 with no security patches on a computer connected to the Internet. Within eight hours of installation, the computer had been attacked. 21 days after installation, the computer had experienced 20 attacks. Approximately 40 days after the installation the computer had been deemed "compromised." Malware can result in consequences ranging from Web site defacement [39] to the loss of human life [6].

Equipped with the knowledge of malware's capabilities, the detection of malware is an area of major concern not only to the research community but also to the general public. Techniques researchers develop for malware detection are realized through the implementation of malware detectors. In this report we are interested in surveying malware detection techniques. Section 2 defines malware. Section 3 gives a general description of malware detectors. Section 4 surveys various malware detection techniques proposed in literature. Finally, Section 5 summarizes this report.

2 What is Malware?

Malware is referred to by numerous names. Examples include malicious software, malicious code (MC) and malcode.

Numerous definitions have been offered to describe malware. For instance, Christodorescu and Jha [11] describe a malware instance as a program whose objective is malevolent. McGraw and Morrisett [35] define malicious code as "any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system." For the purposes of this survey, we adopt the description given by Vasudevan and Yerraballi in [49], which describes malware as "a generic term that encompasses viruses, trojans, spywares and other intrusive code." The canonical examples of malware include viruses, worms, and Trojan horses. This

designation warrants a slightly more detailed discussion on these malware.

Viruses: A computer virus is code that replicates by inserting itself into other programs. A program that a virus has inserted itself into is infected, and is referred to as the virus's host. An important caveat is that viruses, in order to function, require their hosts, that is, a virus needs an existing host program in order to cause harm. For example, in order to get into a computer system, a virus may attach itself to some software utility (e.g. a word processing application). Launching the word processing application could then activate the virus that may, for example, duplicate itself and disable malware detectors enabled on the computer system.

Worms: A computer worm replicates itself by executing its own code independent of any other program. The primary distinction between a virus and a worm is that a worm does not need a host to cause harm. Another distinction between viruses and worms is their propagation model. In general, viruses attempt to spread through programs/files on a single computer system. However, worms spread via network connections with the goal of infecting as many computer systems connected to the network as possible.

Trojan horses: A Trojan horse is malware embedded by its designer in an application or system. The application or system appears to perform some useful function (e.g., give the local weather), but is performing some unauthorized action (e.g., capturing the user's keystrokes and sending this information to a malicious host). Trojan horses are typically associated with accessing and sending unauthorized information from its host. Such Trojan horses can be classified as spyware as well. Malware embedded by its designer is not limited by this kind of malicious activity. The embedded malware could also be a time bomb [28]. For example, malware might create the following scenario; "On May 4, 2010 the compromised system will be programmed to refuse all requests for services."

The sheer number and variety of known and unknown malware is part of the reason why detecting malware is a difficult problem. Christodorescu and Jha [10] and McGraw and Morisett [35] provide detailed descriptions of various types of malware. McGraw and Morrisett note that categorizing malicious code has increasingly become more complex as newer versions appear to be combinations of those that belong to existing categories.

2.1 Who are the Users and Creators of Malware?

Malware writers/users go by a variety of names. Some of the most popular names are black hats, hackers, and crackers. The actual persons or organizations that take on the aforementioned names could be an external/internal threat, a foreign government, or an industrial spy [6].

There are essentially two phases in the lifecycle of software during which malware is inserted. These phases are referred to as the pre-release phase and the post-release phase. An internal threat or insider is generally the only type of hacker capable of inserting malware into software before its release to the end-users. An insider is a trusted developer, typically within an organization, of some software to be deployed to its end users. All other persons or organizations that take on the hacker role insert malware during the post-release phase, which is when the software is available for its intended audience.

In creating new malware, black hats generally employ one or both of the following techniques: obfuscation and behavior addition/modification [11] in order to circumvent malware detectors. Obfuscation attempts to hide the true intentions of malicious code without extending the behaviors exhibited by the malware. Behavior addition/modification effectively creates new malware, although the essence of the malware may not have changed. The widespread use of the aforementioned techniques by malware coders along with those mentioned by researchers [12, 19] suggests that reused code is a major component in the development of new malware. This implication plays a critical role in some of the signature-based malware detection–sometimes referred to as misuse detection–methods as we shall see in Section 4.3.

3 The Malware Detector

As described in the introduction, a malware detector is the implementation of some malware detection technique(s). The malware detector attempts to help protect the system by detecting malicious behavior. The malware detector may or may not reside on the same system it is trying to protect. The malware detector performs its protection through the manifested malware detection technique(s), and serves as an empirical means of evaluating malware detection techniques' detection capabilities.

Malware detectors take two inputs. One input is its knowledge of the malicious be-

havior. In anomaly-based detection, the inverse of this knowledge comes from the learning phase. So theoretically, anomaly-based detection knows what is anomalous behavior based on its knowledge of what is normal. Since anomalous behavior subsumes malicious behavior, some sense of maliciousness is captured by anomaly-based detection. If the malware detector employs a signature-based method, its knowledge of what is malicious comes from its repository, which is usually updated/maintained manually by people who were able to identify the malicious behavior and express it in a form amenable for the signature repository, and ultimately for a machine to read.

The other input that the malware detector must take as input is the program under inspection. Once the malware detector has the knowledge of what is considered malicious behavior (normal behavior) and the program under inspection, it can employ its detection technique to decide if the program is malicious or benign. Although Intrusion Detection Systems (IDS) and malware detectors are sometimes used synonymously, a malware detector is usually only a component of a complete IDS.

4 Malware Detection Techniques

Techniques used for detecting malware can be categorized broadly into two categories: anomaly-based detection and signature-based detection. An anomaly-based detection technique uses its knowledge of what constitutes normal behavior to decide the maliciousness of a program under inspection. A special type of anomaly-based detection is referred to as specification-based detection. Specification-based techniques leverage some specification or rule set of what is valid behavior in order to decide the maliciousness of a program under inspection. Programs violating the specification are considered anomalous and usually, malicious. Signature-based detection uses its characterization of what is known to be malicious to decide the maliciousness of a program under inspection. As one may imagine this characterization or signature of the malicious behavior is the key to a signature-based detection method's effectiveness.

Figure 1 depicts the relationship between the various types of malware detection techniques. Each of the detection techniques can employ one of three different approaches: static, dynamic, or hybrid (see Figure 1). The specific approach or analysis of an anomaly-based or signature-based technique is determined by how the technique gathers information to detect malware. Static analysis uses syntax or structural prop-

erties of the program (static)/process (dynamic) under inspection (PUI) to determine its maliciousness. For example, a static approach to signature-based detection would only leverage structural information (e.g. sequence of bytes) to determine the maliciousness, whereas a dynamic approach will leverage runtime information (e.g. systems seen on the runtime stack) of the PUI. In general, a static approach attempts to detect malware before the program under inspection executes. Conversely, a dynamic approach attempts to detect malicious behavior during program execution or after program execution. There are hybrid techniques that combine the two approaches [38]. In this case, static and dynamic information is used to detect malware.

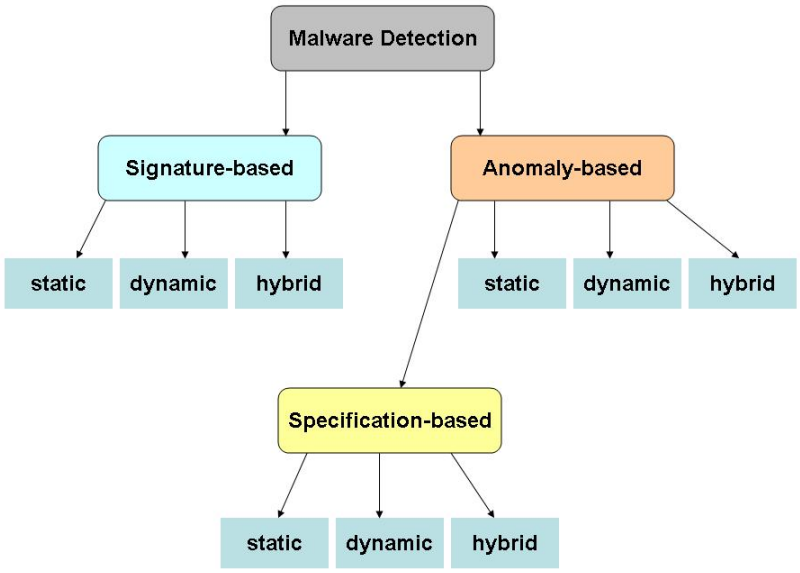

Figure 1: A classification of malware detection techniques.

The remainder of this section provides a description of anomaly-based detection followed by examples from the literature. Even though specification-based detection is a derivative of anomaly-based detection, its prevalence in the literature warrants its own section. Consequently, this section follows the anomaly-based detection section followed by examples from the literature. The specification-based detection section is followed by a description of signature-based detection and examples from the literature.

4.1 Anomaly-based Detection

Anomaly-based detection usually occurs in two phases–a training (learning) phase and a detection (monitoring) phase. During the training phase the detector attempts to learn the normal behavior. The detector could be learning the behavior of the host or the PUI or a combination of both during the training phase. A key advantage of anomaly-based detection is its ability to detect zero-day attacks. Weaver, et al. [53] describe zero-day exploits. Similar to zero-day exploits, zero-day attacks are attacks that are previously unknown to the malware detector. The two fundamental limitations of this technique is its high false alarm rate and the complexity involved in determining what features should be learned in the training phase.

Figure 2 illustrates why anomaly-based detection alone is insufficient for malware detection. As shown, V is the set of all valid behaviors of a system derived from a set of non-conflicting requirements, and V' is the set of all invalid behaviors. As is often the case, an implementation approximates its requirements. Anomaly-based detection attempts to approximate the implementation.

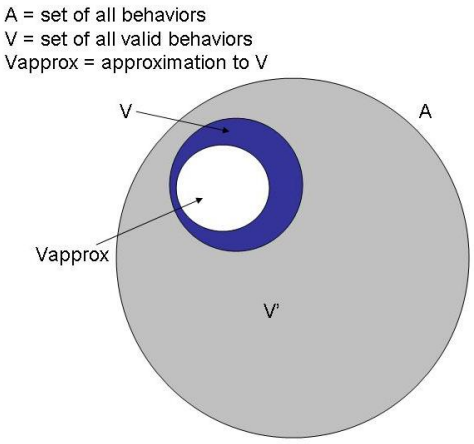

Figure 2: Behavior characterization in anomaly-based detection.

The approximation to all valid behaviors made by anomaly-based detection methods is shown in the figure as the set Vapprox. Since Vapprox is an approximation, valid behavior might be flagged as malicious under anomaly-based detection methods. For example, if an exception is never seen during training phase, an exception seen during the monitoring phase would cause an erroneous alarm. This contributes to

the high false positive rate commonly associated with anomaly-based detection techniques. The possibility for a system to exhibit previously unseen behavior during the detection phase is not zero. Therefore, the probability of an anomaly-based technique raising a false positive is not zero. Developing better approximations to a computer system's normal behavior is an open Computer Science problem.

4.1.1 Dynamic Anomaly-based Detection

In dynamic anomaly-based detection, information gathered from the program's execution is used to detect malicious code. The detection phase monitors the program under inspection during its execution, checking for inconsistencies with what was learned during the training phase.

PAYL

Wang and Stolfo [51] present PAYL, a tool which calculates the expected payload for each service (port) on a system. A byte frequency distribution is created which allows for a centroid model to be developed for each of the host's services. This centroid model is calculated during the learning phase. The detector compares incoming payloads with the centroid model, measuring the Mahalanobis distance between the two. The Mahalanobis distance takes into account not only the mean values of a feature vector but also variance and covariance yielding a stronger statistical measure of similarity. If the incoming payload is too far from the centroid model (a large Mahalanobis distance value), then the payload is considered to be malicious.

Wang and Stolfo evaluated their technique on the 1999 MIT Lincoln Labs data. This data contains 3 weeks of training data, and 2 weeks of testing data. Of the 201 attacks in the Lincoln Labs data, 97 of them should be detected by Wang and Stolfo's technique. The authors' technique could detect 57 of the 97 attacks it should have been able to catch. Overall, the detection rate for their technique was approximately 60 percent when the false positive rate was 1 percent or lower.

Wang and Stolfo also evaluated their technique on a Columbia University Computer Science (CUCS) dataset. The value in running their technique on this data was that it was real data. The MIT Lincoln Labs data, although very thorough, is simulated data. Using real data gives some confidence that the technique is actually effective for use in a real network environment. Due to the privacy policies of Columbia University this

dataset has been destroyed. Consequently, other researchers cannot use this data as another basis of comparison for Intrusion Detectors. PAYL was able to detect the buffer overflow attack of the Code Red II malware in the CUCS dataset.

Data Mining Approaches for Intrusion Detection

Lee and Stolfo [30] propose the use of data mining techniques, namely association rules and frequent episodes, for use in intrusion detection systems. The association rules and frequent episodes can be collectively referred to as a rule set. Rule sets are created for various security critical aspects of the target host (e.g. the frequency with which a combination of system calls is invoked over a small period of time when `rdist` is running). These rule sets serve as knowledge of what is normal for the host's services.

Base detection agents model some specific part of the target system. Meta detection agents use information from multiple base detection agents and output evidence of intrusions. The information takes the form of "audit data." The authors do not give suggestions of what audit data is important to record in order to maximize effectiveness. Detection agents execute rule sets on the audit data received. The authors used `tcpdump` data to learn what was normal for in-coming connections. Through manual examination of normal and abnormal data, the authors were able to deduce that their method could be used to signal the possibility of abnormal activity.

Using Computer Forensic Methods for Privacy-Invasive Software

Boldt and Carlson [4] present a notion of privacy-invasive software (PIS). Adware and spyware are the primary types of privacy-invasive software. Often times, PIS is obtained as a part of file sharing software.

Boldt and Carlson use the Forensic Tool Kit (FTK) to help identify the PIS. The basic approach consists of initially creating a system free of PIS, a "clean" system. A snapshot of the clean system is considered the baseline of the system. The snapshot depicts the file system of the target host. Once the baseline is recorded, some action is performed to potentially release PIS on the target host. The action could be, for example, surfing the World Wide Web. A snapshot would be taken at regular intervals. Ad-Aware was the most popular PIS removal tool, and so the authors chose to assess Ad-Aware using forensic and static analysis techniques. Through using their technique, Boldt and Carlson found that Ad-Aware produced false positives as well as

false negatives.

Short Sequences of System Calls

Hofmeyr et al. [21] propose a technique that monitors system call sequences in order to detect maliciousness. First, profiles must be developed that represent the normal behavior of the system's services. "Normal" in this technique is defined in terms of short sequences of system calls. Although intrusions may be based on other parameters, these other parameters are ignored. Hamming distance is used to determine how closely a system call sequence resembles another. A threshold must be set to determine whether a process is anomalous. Typically, processes showing large Hamming distance values are anomalous. Hofmeyr et al.'s method was able to find intrusions that attempted to exploit various UNIX programs like `sendmail`, `lpr`, and `ftpd`.

FSA for Detecting Anomalous Programs

Sekar et al. [42] created a Finite State Automata (FSA) based approach to anomaly detection. Each node in the FSA represents a state (program counter) in the PUI which the algorithm utilizes to learn normal data faster and perform better detection. Transitions in the FSA are given by system calls.

In order to construct the FSA, the program is executed multiple times. When a system call is invoked, a new transition is added to the automaton. The automaton resulting from the multiple executions will be what the algorithm considers normal. During runtime, system calls are intercepted, and the program's state recorded. If an error occurs in doing this, then an anomaly has occurred. Next, the algorithm checks for a valid transition from the FSA's current state to the newly invoked system call. If no such transition exists then there is an anomaly. If the previous two steps were successful, then the FSA is transitioned to the next state.

Sekar et al. compare their approach to the n-gram approach proposed by Hofmeyr et al.'s work in [21]. Sekar et al. implemented the n-gram approach and compared their FSA approach to that implementation. They evaluated the two approaches on `httpd`, `ftpd`, and `nsfd`. The FSA method was found to have a lower false-positive rate when compared with the n-gram approach. However, there is no clear indication of whether optimal parameters were chosen for the n-gram approach. For example, a key parameter that will affect detection ability in the n-gram approach is its threshold value. In addition to their results, the authors assert without empirical evidence that

their technique is robust against buffer overflow attacks, Trojan horses, maliciously crafted input, password guessing attacks, and Denial-of-Service attacks.

Process Profiling System Calls

Sato et al. [40] proposed a detection method based on the frequency of system calls. A process's profile is composed of a sequence of system calls. The base profile of a process has system calls ranked based on frequency. The lower the system call's ranking number, the more frequent the system call is used by the described process.

An SUID records system call sequences for daemons. Once the process's profile has been established, the distance between the profile and sample data must be measured. Sample data is the program under inspection's runtime data. An optimal matching algorithm called DP matching is used to obtain the distances between the sample and profile data. The distance given by DP matching serves as an indicator of a process's maliciousness when the distance is compared to the number of system calls available in the system. If the distance of a process's system call trace and its profile is less than 1/4 of the total number of system calls in the system then it is in a normal state. If the distance is greater than 1/4 and less than 1/2 then the process is considered to be in a caution state. If the distance is greater than 1/2 then the process is considered to be in a detect state.

As previously mentioned, system calls are represented by their rank derived from the base profile. The authors justified their frequency ranking approach by comparing it to two other ways of identifying system calls. The alternatives were assigning the system calls their actual system call number or assigning the system call numbers randomly. The approach which modeled normal behavior most accurately was the rank by frequency method chosen by the authors. Using this method combined with DP Matching, `rlogin` and `portscan` intrusion attempts were detected successfully. However, the authors method was unable to detect intrusions on `Qpopper` and `eject`.

Variable Length Audit Trail Patterns

Wespi et al. [54] propose a detection method using varied length audit trail patterns. The audited events of a process constitute a pattern. A pattern must have at least 2 events and occur at least twice. A maximal pattern is not a subsequence of another pattern, and occurs most frequently. The algorithm processes sequences from the beginning to the end. If there are few matches between the known patterns for a

process and the corresponding process under inspection, then there is a high likelihood that the process under inspection is a possible intrusion.

Wepsi et al. evaluate their method using the `ftpd` process. They compare their method to Hofmeyr et al.'s [21] comparable method which uses fixed length audit trail patterns. From the `ftpd` process, 65 unique benign sequences were derived from it. 17 percent of the fixed length patterns matched these benign sequences of the `ftpd` process. 72 percent of the benign sequences were matched with the variable length approach. Based on this experiment, the variable-length approach appeared to be significantly more accurate than the fixed length approach.

NATE

Taylor and Alves-Foss [48] propose a computationally low cost approach to detecting anomalous traffic. Their approach is referred to as the Network Analysis of Anomalous Traffic Events (NATE). This technique focuses on attacks which exploit network protocol vulnerabilities. Their approach relies on the assumption that malicious packets tend to have a large number of `syn`, `fin`, and `reset` packets, while having a low number of `ack` packets. This approach also relies on abnormalities showing up in the number bytes being transferred for each packet. A session is defined as information flow from a source to some IP address and port number pair. Multivariate cluster analysis is used to group the normal TCP/IP sessions.

To evaluate their technique Taylor and Alves-Foss used the 1999 MIT Lincoln labs data, namely the FTP, HTTP, and SMTP data. The authors noted that the drawback in using this data set is that it is simulated, and therefore, their tool, NATE, may behave differently in a real environment. They used Mahalanobis distance to measure the distance between the known attacks and the normal clusters. Portsweep, Satan, and Neptune were significantly distant from the normal clusters and consequently easily found to be anomalous. However, the Mailbomb seemed to match some of the generated normal clusters.

Specification and Anomaly Detection Approach to Network Intrusion

Although Sekar et al. [44] consider that their detection method as both specification-based and anomaly-based, given how we have defined specification-based and anomaly-based detection in this report, this method would be more aptly named "A Model-based Approach to Anomaly Detection." Network behavior is modeled with an EFSA, which

is an extended finite state automaton. An EFSA can (1) make transitions on events that have arguments, and (2) use a finite set of state variables in which values can be stored. EFSAs model the network interface of the gateway host of the target network.

Sekar et al. leverage statistical properties seen in network traffic to determine the maliciousness of the network events on a target network. For example, the number of timeout transitions that are taken over some subset traces of the EFSA can be taken to identify useful properties about the data stream. Another example would be to analyze the distribution of state variable values seen over some period of time. Anomalous behavior is based on repetition. The authors use the Lincoln Labs 1999 evaluation data. Sekar et al.'s approach were able to detect all attacks in the 1999 data that were within the scope of their method. Their method generated 5.5 false alarms a day, which is low when compared to false alarm rate reported in the Lincoln Labs evaluation data.

4.1.2 Static Anomaly-based Detection

In static anomaly-based detection, characteristics about the file structure of the program under inspection are used to detect malicious code. A key advantage of static anomaly-based detection is that its use may make it possible to detect malware without having to allow the malware carrying program execute on the host system.

Fileprint Analysis

Li et al. [31] describe Fileprint (n-gram) analysis as a means for detecting malware. During the training phase, a model or set of models are derived that attempt to characterize the various file types on a system based on their structural (byte) composition. These models are derived from learning the file types the system intends to handle. The authors' premise is that benign files have predictable regular byte compositions for their respective types. So for instance, benign .pdf files have a unique byte distribution that is different from .exe or .doc files. Any file under inspection that is deemed to vary "too greatly" from the given model or set of models, is marked as suspicious. These suspicious files are marked for further inspection by some other mechanism or decider to determine whether it is actually malicious.

Li et al. found that applying 1-gram analysis to PDF files embedded with malware pretty effectively relative to the COTS AV scanner they compared their technique to in their experiments. Li et al.'s technique exhibited detection rates between 72.1 percent

and 94.5 percent for PDF file that had embedded malware, whereas the COTS AV scanner had a detection rate of zero effectively. The caveat of the aforementioned experiment is that the embedded malware is embedded in the either the head or tail of the PDF files tested. Since it is also possible to embed malware in the middle of a PDF file carefully, such that the reader may still be able to open the file, the authors found it worthwhile to assess their techniques ability to detect malware in PDF files that have malware embedded in the middle portion of PDF files. The authors believe that more work needs to be done to determine the viability and effectiveness of 2-gram or 3-gram analysis. When tested on different file types, the detection results varied.

4.1.3 Hybrid Anomaly-based Detection

Strider GhostBuster

Wang et al. [52] propose a method for detecting a type of malware they refer to as "ghostware." Ghostware is malware that attempts to hide its existence from the Operating System's querying utilities. This is typically done by intercepting the results for these queries and modifying them so traces of the ghostware could not be found/detected via API queries. For example, if a user performs a command to list the files in the current directory, "dir," the ghostware would remove any of its resources from the results returned by the "dir" command.

Wang et al. offer a "cross-view diff-based" approach to detecting these type of malware. In addition to this approach they offer two ways of scanning for the malware, one being an inside-the-box approach and the other an outside-the-box approach. Since there are many layers that return values, and actual arguments must pass through them when a system call is made, many opportunities are afforded to ghostware to intercept function calls. The authors' proposed method to counter this vulnerability will compare the results from a high-level system call like "dir" to a low-level access of the same data without using a system call. An example of a low-level access may be accessing the Master File Table (MFT) directly. This described process is considered the "cross-view diff-based" approach.

The inside-the-box approach mandates that the comparison of the high-level and low-level results are within the same machine. However, one may imagine that the ghostware may compromise the entire Operating System in which case the low-level

scan cannot be trusted. Hence, an alternative to the inside-the-box approach is the outside-the-box approach.

In the outside-the-box approach, another (clean) host performs the low-level access without the target host's knowledge. The high-level scan of the target host is compared to the low-level scan from the clean host. If there is any difference between the low-level or high-level scans in either the inside-the-box or outside-the-box approach then ghostware is present in the target host.

In detecting file-hiding ghostware (e.g. ProBot SE and Aphex), the inside-the-box approach did not produce any false positives. However, the outside-the-box approach did produce some false positives for the 10 ghostware the authors used in their experiments. For registry-hiding ghostware (e.g. Hacker Defender 1.0 and Vanquish), virtually no valid false positives were found. The one false positive found, over the six ghostware used in this experiment, was fixed with a minor change. For process/module-hiding ghostware (e.g. Berbew and FU), no false positives were found for the four ghostware used in the experiment. The authors do acknowledge that it is possible for false positives to occur for these type of ghostware, but did not see any during experimentation.

Self-Nonself

Forrest et al.'s technique in [17] is generally infeasible for real detection tools. The goal of the proposed technique is to detect modifications to data being protected. A caveat of this approach is that it cannot detect the removal of items from the protected data collection. "Self" is defined as the protected data. "Other" is all data that do no match "Self." It may becoming clear why this may be infeasible. In order to be effective, Other must be approximated. The approximation proposed by the authors still results in unpalatable computational costs. The idea is that if Self is modified, a match will be made with an element from the Other collection. As the probability of matching two random data (strings) exactly for arbitrarily sized data (strings) is extremely low, the authors relax the notion of matching. The authors approach allow for matching to be defined by the user. For example, if comparing strings of size ten each, a match can be defined as a contiguous subsequence of length two starting at the same position in both strings.

In some of the tests conducted, the authors created infected files by mutating a single character in a file. The results showed that with more detectors available, the

detection of the this modification is discovered at a higher rate. They evaluated their technique on a real virus, namely, the TIMID virus. The results show that for 2 detectors, the virus can be detected 76 percent of the time reliable. When 10 detectors are present, the virus is found virtually all the time.

4.2 Specification-based Detection

Specification-based detection is a type of anomaly-based detection that tries to address the typical high false alarm rate associated with most anomaly-based detection techniques. Since specification-based detection is a derivative of anomaly-based detection Figure 2 is also valid for specification-based detection. Instead of attempting to approximate the implementation of an application or system, specification-based detection attempts to approximate the requirements for an application or system. In specification-based detection, the training phase is the attainment of some rule set, which specifies all the valid behavior any program can exhibit for the system being protected or the program under inspection. The main limitation of specification-based detection is that it is often difficult to specify completely and accurately the entire set of valid behaviors a system should exhibit. One can imagine that even for a moderately complex system, the complete and accurate specification of its valid behaviors can be intractable. Even when it may be straight-forward to express specifications for a system in natural language, it is often times difficult to express this in a form amenable for a machine.

4.2.1 Dynamic Specification-based Detection

Approaches classified as dynamic specification-based use behavior observed at runtime to determine the maliciousness of an executable.

Monitoring Security-Critical Programs

Ko et al. [25] propose a specification-based method for detecting maliciousness in a distributed environment. An implementor would specify the trace policy for the system. A trace is simply an ordered sequence of execution events, which are essentially the system calls recorded by the auditing mechanism. Ko et al. created a parallel environment (PE) grammar to address synchronization issues in distributed programs. Audit trails are parsed in real time.

The authors developed an implementation of their approach called Distributed Program Execution Monitor (DPEM). Trace policies were created for 15 Unix programs. One trace policy was created for the `rdist` program. DPEM was able to catch two violations present in an attack on the `rdist` program. The violation signal occurred approximately .06 seconds after the actual violation. The authors observed similar delay when they created a trace policy involving `passwd` and `vi`. Intrusion attempts on `sendmail` and `binmail` were also detected by DPEM in .1 seconds.

Using Dynamic Information Flow to Protect Applications

Masri et al. [34] describe a tool called the Dynamic Information Flow Analysis (DIFA). This tool was designed specifically for Java applications. This tool has the ability to monitor method calls at runtime by instrumenting the bytecode classes of applications. Hence, information flow can be captured each time a method is invoked by the application and can be compared against known information flow policy. For example, certain directories may be declared, by the policy creator, to be "SensitiveSources," which are objects that may be involved in an illegal information flow. A policy may exist that prohibits certain user/applications from writing this information. This policy would be verified in a "sink" method. The sink methods in this scenario would be any output function invoked by the unauthorized user/applications like `write()` or `send()`. This technique can also be used for known malicious information flows, which would then allow this tool to be used as a signature-based tool instead of a specification-based tool.

Masri et al. give a case study to show the functioning of their DIFA tool. The case study involved executing a security exploit on the Java application server, Apache Tomcat. After instrumenting Apache Tomcat, the authors sent normal (benign) requests and attack requests to the server. Based on the clusters produced from the 150 requests sent to the Apache Tomcat server, it appears that anomalous request could be detected from the plotted information flow profiles of each requests. When the profiles are plotted on a graph, the normal request profiles are in different partitions of the graph.

ACT: Attachment Chain Tracing

Xiong [55] proposes a method to detect malicious hosts in a network. This method's goal is to thwart malware that spread via email attachments. The ACT method is

modeled after classical epidemiological notions. An epidemiological link exists between two hosts, *A* and *B*, if host *A* sends an email with an attachment to host *B*. A node that sends a host, *i*, an email with an attachment is considered to be at layer one. Put another way, if a node i has node *z*'s email address, then node *z* is considered to be at layer one. If an epidemiological link exists between a host at layer one and another host, but not with host *i*, then that host is considered to be at layer two. Subsequent layers follow similarly.

In order to identify nodes that are suspicious, we must first set a threshold on the number of emails a host sends out. The detection interval should be based on the normal duration for an attachment of an email to be downloaded from the server. Then the system administrator would decide the number of layers that would have to be identified from any host *i* before making the decision of what hosts were infected. For example, if the system administrator chooses three layers to be the infection deciding point, then hosts, with respect to some host *i*, at each layer would have to be declared suspicious before all hosts would be decided to be infected. The host *i* is determined by finding the host that appears to be infected but has no epidemiological links connected to it.

Xiong's method was evaluated on two different network topologies because there is no agreement on the email network topology. One topology used was the random network topology and another network topology used was based on power law. For the non-reinfection case, Xiong's simulation shows that ACT was able to completely stop virus propagation in the random network topology. ACT was able to keep the number of infected hosts to a low percentage of the total hosts in the power law network topology. When reinfection is allowed, which is more realistic, ACT does not perform very well at detecting and quarantining infected hosts. The infection spreads approximately at exponential rates in both topologies.

Automated Detection of Vulnerabilities in Privilege Programs

Ko et al. [24] present a specification language for specifying the intended behavior of privileged programs. This technique relies on the Operating System to generate audit trails that are then used in the validation of program behavior.

Auditing is the process of logging "interesting" activities, which in this case is whenever a system call is invoked. Based on a program's specification, its runtime behavior can be decided to be malicious or not. The program's specification is translated into

audit-trails that is compared to the audit-trails of the PUI. The audit-trails of the PUI are captured by the Operating System.

A potential disadvantage of this technique is that the malware would be detected after the attack. Another potential disadvantage is that the technique can only be as granular as the Operating System's auditing mechanism. No empirical study of the effectiveness of this approach was given.

Process Behavior Monitoring

Sekar et al. [43] propose a method where a program is manually translated into an Auditing Specification Language (ASL). Next, ASL code is compiled into a C++ class. This C++ class is then compiled and linked with an infrastructure that captures system calls. This creates an executable that is referred to as the system call detection engine. Each system call invoked at run time is captured and sent to the system call detection engine. The system call is intercepted just before and just after performing the kernel level functions or behaviors. A detection engine compares the system call being made against the specification initially modeled by the ASL specification of the PUI. No empirical evidence is given in this work.

Enlisting Hardware in Malicious Code Injection

Lee et al. [29] suggest that a processor can accurately identify valid call and return instructions by leveraging the LIFO nature of the procedure call stack. This method is designed to be robust against stack smashing attacks. The processor would maintain its own stack, called a Secure Return Address Stack (SRAS) which would also be a LIFO data structure. With a SRAS at its disposal, the processor can identify attacks based on whether the stack in memory is consistent with the SRAS. In order for this method to work, the SRAS must be placed in a secure place by the kernel that would prevent it from being modified by malware. In general, call and return addresses occur in a LIFO fashion, though this is not necessarily always the case.

Lee et al. depict times where it is appropriate behavior for an application to cause the stack pointer to jump to a deeper point in the stack (and consequently not follow the LIFO nature of the procedure call stack). An example offered by the authors is exception handling in C++. In order to make Lee et al.'s technique viable in the presence of exception handling in C++, the authors offer a few ways for dealing with exception handling. One option is to not allow non-LIFO behavior. Another option is to allow

users to disable SRAS at will. Another alternative would be to force the user to recompile the application, and only allow certain types of non-LIFO behavior, namely going deeper into the stack. Another alternative, would not involve recompiling the application, and would involve dynamically inserting SRAS push and pop instructions into the executable for the known non-LIFO procedures. Lee et al.'s simulation revealed that their technique had negligible performance degradation consequences (less than 1 percent for SRAS of 128 entries). They evaluated their technique on 12 SPEC2000 integer benchmarks. There was one benchmark (`parser`) where the SRAS has 64 entries where the performance degradation 2.11 percent.

Mitigating XSS Attacks from the Client-Side

Kirda et al. [23] propose a technique for preventing cross-site scripting (XSS) attacks that attempt to compromise a user's personal information. Their implementation is called Noxes, which is a client-side tool that is to be used with the end user's browser.

Noxes is modeled after contemporary personal firewalls. It monitors all connections made by the user's browser. Noxes performs a few optimizations to make the tool more viable for practical use. For example, it considers all embedded static links as safe relative to the type of XSS attacks Noxes is designed to prevent. It also considers links from the user's local host to be safe as well. If by visiting a malicious site, an executable is generated that attempts to make a connection to a Web site which is not found in the rule set, then this action will be disallowed. Temporary rules are made for the statically embedded links on the Web sites that do not appear in the user's rule set. If statically embedded links which were saved are not visited for some period of time, they are deleted from the user's rule set.

The authors created a web crawler to see the effect of different threshold values for the number of links a Web page can have to the same domain. The web crawler crawled 800 Web sites and 110,00 distinct web pages. The claim is that connection alerts would not be generated for too many of the Web pages traversed (5.7 percent). However, there is no indication of the connection alerts' accuracy. In other words, are these alerts valid, or are they false positives/negatives. In the evaluation of Noxes, Kirda et al. state that their tool is intended for the "sophisticated" user.

Dynamic Information Flow Tracking

Suh et al. in [45] propose a method which would require modification to the Oper-

ating System and processor. Their method is based on the notion that the data can be marked as either safe or unsafe–authentic or spurious, respectively.

A security policy specified by the Operating System will identify the spurious data. Spurious data are typically all the privileged I/O of networking mechanisms. The Operating System does this by setting a bit of the identified data. The processor ensures that the control in the program is not contingent on the spurious data. Spurious data should not be a jump target nor should it be the address of a load/store operation–unless, however, it is within a known bound that has been explicitly checked. If it finds that control is based on spurious data–by checking the bit that is tagged by the Operating System–the processor will throw an exception that is caught by the Operating System. At the point the Operating System catches this exception, the action typically taken by the Operating System is to kill the process that caused the exception.

The data is tracked through dependencies, that is, if any data created that was dependent on some spurious data, then it too is spurious. For example, if one operand of the `add` operator is marked as spurious while the other is not marked as spurious, then the resultant will be marked as spurious.

The authors evaluated their approach against stack buffer overflows, heap buffer overflows, vudo (heap buffer overflow), and format string attacks. Their approach was able to stop all attacks with no false alarms.

Using Instruction Block Signatures

Milenkovic et al. [36] propose a technique that would dedicate some of the processor's resources to ensure only secure instructions are executed. In this scheme, instruction block signatures are verified at runtime. Instruction block signatures are encrypted by a secret processor key unique to each processor.

Milenkovic et al. implemented three variations of their technique, namely, SIGT, SIGE, and SIGC. In SIGT and SIGE, instruction blocks correspond to basic blocks in the code, with each basic block having a signature associated with it. The differences in the three approaches lie in how and/or where the signatures are stored. The signatures for the basic blocks are determined via a function called the Multiple Input Signature Register (MISR).

In general, for the various implementations, the assumption is that injected malicious code will probably change the control flow of an executable. If the signature for the basic block cannot be found–because, for instance, the basic block is a block of in-

jected code–then the basic block under inspection is considered malicious. Milenkovic et al. determined that their technique did not decrease system efficiency significantly, and consequently suggests that their technique is viable.

Fast Detection of Scanning Worms

Schechter et al. [41] propose reverse sequential hypothesis testing as a method for detecting worms. The method rests on the assumption that scanning worms tend to reside on hosts that have low successful connection rates. The proposed method requires at least ten observations to make a decision about a host. Thresholds are used to indicate whether a host is clean or infected. For each, unsuccessful connection a host is considered closer to an infected threshold and is moved further from the clean threshold. At each connection outcome, a new sequential hypothesis is run in the opposite direction to capture hosts that may not have been infected at the time of inspection, but became infected.

A credit based connection rate limiting method is instituted to aid the sequential hypothesis method in the detection of scanning worms. In the credit based approach, each host starts with 10 credits. For each successful connection, a host is credited with 2 credits, and 1 credit is subtracted for each failed attempt. If a host's connection request drops to 0 then connection requests from this host will be blocked. Peer-to-Peer applications appear to show "ambiguous" behavior and are placed in their own separate category. Since the authors' method requires 10 observations, attacks successful before reaching 10 will be able to proceed undetected.

Schechter et al.'s method does have the ability to catch worms that scan machines slowly provided they attempt to connect to 10 hosts. The method will not catch topological worms which are worms that contact hosts it has already contacted, flash worms, which use attack lists from previous scans or colluding worms. Colluding worms typically exist in different networks and give the illusion that the scanning worms in a network have a high success rate. Another way of increasing its success rate is for the worm to communicate with well known services that it knows it will get a reply from. IP masquerading is also a concern, because this could lead to the wrong host being quarantined.

Schechter et al.'s algorithm was evaluated on real data taken from a medium sized ISP. The data was taken at two points, once in April of 2003(with 404 local active hosts) and again in January of 2004 (with 451 local active hosts). The duration for April and

January were 627 minutes and 66 minutes respectively. The authors compared their reverse sequential hypothesis testing to their implementation of virus throttling. Virus throttling is a mechanism used to reduce the number of outgoing first-contact connections from a host. The basic idea of virus throttling is to queue outgoing first-contact connections when the current working set is beyond some threshold. For example if the threshold is 5, and the host has made 5 first-contact connections with other hosts, subsequent first-contact connections are queued. Then once per second, the least recently used first-contact connection in the working set is removed, and a queued first-contact connection is enqueued and placed in the current working set. The authors' algorithm, though operationally slower than virus throttling, outperformed virus throttling's effectiveness by more than two fold.

Protecting Against Unexpected System Calls

Linn et al. [32] propose a method where a new section is added to the ELF (Executable and Linkable Format) binaries. This new section is the IAT (Interrupt Address Table) section. The IAT contains the system call number corresponding to the system call appearing in the executable as well as the address after the system call. The Linux kernel would be modified so that if an executable has an IAT section then this information would be stored in the process's data structure. If the system calls made at runtime are inconsistent with IAT found in the processes data structure a signal is raised.

Linn et al. also leverage address obfuscation to increase the difficulty level for an attacker to successfully compromise an executable. This method is effective in preventing scan attacks which look for the 2-byte sequence "0xcd80," which is the encoding for the "int x80" instruction.

Synthetic attacks were used in the Linn et al. experiment to simulate attacks their approach is concerned with stopping. Since the goal of the authors' approach is to stop malicious code from executing system calls, the means by which the malicious code gets into an application is irrelevant with respect to Linn et al.'s technique, and consequently their experiments. The authors used synthetic attacks to evaluate their method because, in their opinion, using only known attacks cannot give a true sense of an IDS's effectiveness. In their evaluation studies, the authors found that their method correctly identified injected system call instructions, i.e. system calls made from the stack or heap, and were successful against scan attacks. All synthetic attacks failed to circumvent the authors' proposed detection method.

4.2.2 Static Specification-based Detection

During the detection phase, static specification-based detection uses the structural properties of the PUI to determine its maliciousness.

Static Detection of Malicious Code in Executables

Bergeron, et al. [2] propose a method that attempts to analyze the malicious intent of an executable before it is executed. This form of malware detection makes use of static analysis to identify the malicious code. First, in order to obtain a better understanding of what the executable is doing, the binary executable is disassembled into an abstract intermediate representation. This intermediate representation is assembly code. The assembly code is then parsed to generate a syntax tree from which a control flow graph is created. From this control graph an API-graph is created where only API calls are kept while all other computations are not depicted. From this API-graph a critical-API graph is created where the user determines what API calls are critical through the use of security policies which are represented as an automaton, also referred to as security automaton.

Transitions in a security automaton occur when the host system performs any action, more specifically those actions of the PUI. At least one of the states of the security automaton is to be identified as the bad state. For example, a security automaton may specify that after the PUI reads any sensitive file, no network utilities can be used by the process, such as, `send()`. If the system enters a bad state based on the program's critical-API graph, then the system is in violation of the security policy specified by the automaton, and the executable is deemed malicious. Hence the critical-API graph is checked against this security automaton to determine the existence of malicious behavior. A sample malicious program the authors successfully tested their approach on is the WINIPX.EXE program.

Static Analysis of Binaries

Bergeron et al. [3] propose a specification-based detection method. First the code is disassembled into assembly code. Then to make the assembly code easier to analyze, it is transformed into a higher level representation. Flow analysis is used to help create the high level abstraction. Suspicious APIs of the PUI are identified based on the behavioral specification of the system. Program slicing would then be used to identify critical regions of the code (especially those involving API calls) that are to be checked

against the behavioral specifications while simultaneously making the problem of detection easier. If the specifications are violated, then the program under inspection is deemed malicious. Bergeron et al. give no empirical study for their proposed method.

This work differs from [2] primarily in its specification of how to derive a high-level imperative representation and its use of program slicing. For example, in order to make the disassembled code easier to analyze, the program stack is eliminated. That is all `pop()` and `push()` calls are eliminated and replaced with `mov()` instructions that move values in to or out of temporary variables. Program slicing produces a subset of program statements considered security relevant when given a slicing criterion (a node from the CFG and a subset of the program variables).

Compiler Approach to Malcode Detection

Debbabi et al. [14] propose a compiler based approach to ensuring the security of code. A certifying compiler takes source code as input and creates binary code as well as annotations that consist of the types for the assembly code of the source code. The annotations also include information pertaining to the behavior of the assembly code as it pertains to security. An example of a security relevant behavior would be accessing sensitive or private resources. A component referred to as the verifier ensures that the types in the certificate agree with those found in the binary executable. Then based on security policies, the verifier decides whether the executable is secure. If the executable is not deemed to be secure, it is not allowed to execute. No results are reported for this work.

Detecting Malcode in Firmware

Adelstein et al. in [1] propose a method where malicious boot firmware that initializes hardware and loads the Operating System is the target. This is an area of vulnerability because it is code that is executed before the OS is loaded. An untrusted firmware module is verified against a security policy prior to loading it into memory. In general, these security policies identify how device drivers are allowed to interface with the rest of the system.

A certifying compiler compiles and annotates the untrusted firmware modules. In this study the compiler accepted Java bytecode. The output of the certifying compiler is a representation amenable for the verifier to determine if the firmware module can be trusted. Their method is based on Efficient Code Certification that ensures the

following: control flow safety, memory safety, and stack safety. At the time of the paper, the authors were still developing the implementation, so no preliminary results were given. The ideas given in this appear are very similar to that given in [14], however no comparison is given of the two approaches.

4.2.3 Hybrid Specification-based Detection

DOME

Rabek, et al. [38] offer a technique called DOME (Detection Of Malicious Executables). DOME was designed to detect injected, dynamically generated, and obfuscated code. DOME is characterized by two steps. In the first step, DOME statically preprocesses the PUI. Preprocessing consists of (1) saving system call addresses, (2) their names, and (3) the address of the instruction directly following each system call. The third component saved by DOME are the return addresses for system calls in the executable. In the second step, DOME monitors the executable at runtime, ensuring that all system calls made at runtime match those recorded from the static analysis performed in the first step. The API is instrumented with pre-stub and optionally post-stub code at load time. The pre-stub code ensures that items 1 - 3 from the preprocessing stage match what is seen at execution time. In the proof of concept study conducted by Rabek et al. they found that DOME was able to detect all system calls made by the malicious code.

Intrusion Detection via Static Analysis

Wagner and Dean [50] propose a technique which analyzes the source code of a program and derives a CFG that represents its system call trace. Thus an alarm is triggered if it is found that during execution a system call was made that was not in the model. This type of model is referred to as the call graph model. The call graph model proposed here does, however, allow infeasible paths. For example, in this model it may be valid for a function call to be made and have it return to an address other than its return address. To combat this, an abstract stack model was proposed which is essentially a virtual procedure call stack. Maintaining this virtual call stack model may be expensive computationally, depending on its implementation.

Wagner and Dean also proposed a digraph model where a sequence of two system calls would be used to determine whether or not a system call trace was malicious. In

general this approach in intractable. Wagner and Dean use branching factor to help evaluate their approach. If an application's execution were to be frozen at some point in its execution, then the branching factor would be the set of system calls that would be allowed to execute next without setting off any alarms. Having a small branching factor is desirable as this suggests that attackers who try to circumvent IDSs by mimicking the normal behavior of the PUI (mimicry attack) will have fewer ways to successfully attack a system without being detected. Wagner and Dean found that checking the arguments to system calls helped with performance and precision. This would suggest that it is always desirable to check the arguments to system calls as it makes the models more precise and decreases the number of valid possible paths for a given model and consequently, a given execution. In evaluating their models, the authors found that generally the abstract stack model was most precise, then the call graph model, and then the digraph model.

Detecting Manipulated Remote Call Streams

Giffin et al. [18] propose a technique to protect host machines that have processes that execute remotely and send requests back to the host for the execution of malicious sequences of system calls, often times referred to as a malicious remote stream. The technique is comprised of two steps. The first step is to perform static analysis on the executable before execution to derive a CFG (Control Flow Graph) which contains all possible remote streams for the executable. As the process executes remotely, the local agent checks each request and ensures the call remains in the modeled NFA/PDA (derived from the executable's CFG). If a request ends up not being in the model, then the stream is considered manipulated.

The ability of the technique to detect manipulated remote streams, relies heavily on the model's precision. One way to increase the model's precision, employed by the authors, was to get rid of dead automata. They also employed automata inlining and null call insertion to help get rid of infeasible paths in their automata. A possible drawback of this technique is that it could require considerable bandwidth to be effective. This arises in null call insertion. Each call goes through the checking agent which must be done over the network. This increases the number of kernel traps as well as adds to the network traffic. This approach was found to be more efficient than Wagner and Dean's approach [50], however, Giffin et al. found it difficult to compare the precision of the two approaches as their implementations were done on two different systems.

Wagner and Dean implemented their system on Red Hat Linux, whereas Giffin et al. used Solaris 8.

Preventing SQL Injection Attacks

Halfond and Orso [20] present an implementation, Analysis and Monitoring for NEutralizing SQL-Injection Attacks (AMNESIA), designed to prevent SQL injection attacks. Their implementation leverages static analysis to identify the hotspots in the Web application that accept user input and uses that input to generate an SQL statement to access the database.

Through static analysis, a non-deterministic finite automaton (NDFA) is derived for each hotspot to identify all valid statements for the given hotspot. The Web application is monitored at runtime. Before allowing the flow of control to reach a hotspot, the statement is validated against the NDFA. If the statement is validated, then the SQL statement is allowed to execute. Otherwise the user's input is deemed malicious and an exception is thrown specifying some details of the attack.

AMNESIA protected seven Web applications from thousands of attacks. It produced no false positives or false negatives, although there is potential for both in using this method, as the set of valid statements is a proper subset of the possible statements generated by the NFDA.

StackGuard

StackGuard [13] prevents a type of buffer overflow called "stack smashing." StackGuard is a compiler extension that can detect the change of active return addresses or simply prevent writing to active return addresses for programs compiled with StackGuard. Detection of the buffer overflow attack exploits the fact that stack smashing is a linear attack. In other words, typically, stack smashing overwrites everything up to and including the return address. So StackGuard inserts a "canary word" close to the return address. Before going to the return address, the canary word is checked. If the canary word has changed then control is not transferred to the return address. In order to prevent overwriting of a return address, StackGuard leverages MemGuard which is a tool for offering memory protection. When a function is active (is called) its return address is made read-only (via its virtual page), and making the location writeable after the function returns. In all experiments conducted by the authors, StackGuard was able to detect the stack smashing attacks.

SPiKE

SPiKE, [49], is a framework designed to help users monitor the behavior of applications, with the goal of finding malicious behavior. The monitoring is done by instrumenting Operating System services (system calls). The authors claim that "most if not all malware are sensitive to code modification." If this statement is true, then in order for binary instrumentation to be useful, it must be more stealthy to malware. For example, instrumentation introduces abnormal latency for a system call. Malware may request for the real-time clock time and ascertain that it is being monitored because system calls are taking too long to complete. SPiKE combats this or hides itself by applying a clock patch such that any requests will resemble a time closer to what the malware would expect (i.e. the real-time clock would reflect a time that would be consistent with the time generated by the real-time clock had normal execution of the system call taken place).

SPiKE allows for instrumentation anywhere in an executable with the use of "drifters." A drifter can be described by its 2 components: a code-breakpoint and an instrument. Code-breakpoints are implemented by setting the "not-present" attribute of the to-be-instrumented memory location's virtual page. Once the page fault exception is raised, an opportunity for stealth instrumentation avails itself. The instrument component of a drifter is simply the monitoring (or whatever functionality) the user desires to perform. In the assessment of Vasudevan and Yerraballi, SPiKE appeared to successfully track the W32.MyDoom Trojan, which is an intelligent malware instance that can identify traditional binary instrumentation.

4.3 Signature-based detection

Signature-based detection attempts to model the malicious behavior of malware and uses this model in the detection of malware. The collection of all of these models represent signature-based detection's knowledge. This model of malicious behavior is often referred to as the signature.

Ideally, a signature should be able to identify any malware exhibiting the malicious behavior specified by the signature. Like any data that exists in large quantities which requires storage, signatures require a repository. This repository represents all of the knowledge the signature-based method has, as it pertains to malware detection. The repository is searched when the method attempts to assess whether the PUI contains

a known signature.

Currently, we primarily rely on human expertise in creating the signatures that represent the malicious behavior exhibited by programs. Once a signature has been created, it is added to the signature-based method's knowledge (i.e. repository). One of the major drawbacks of the signature-based method for malware detection is that it cannot detect zero-day attacks, that is an attack for which there is no corresponding signature stored in the repository.

Figure 3 illustrates the major disadvantage of signature-based methods. Since the set of possible malicious behaviors, U, is infinitely large, there are no known techniques for accurately representing U via signatures. Furthermore, a repository of signatures is a weak approximation to U. Another drawback of signature-based methods is that human involvement/expertise is usually needed to develop the signatures. This not only allows for the introduction of human error, but takes considerably more time than if signature development was completely automated. Given that some malware has the capability to spread extremely fast, the capability to quickly develop an accurate signature becomes paramount. Automated signature builders do exist [26], but more work needs to be done in this area.

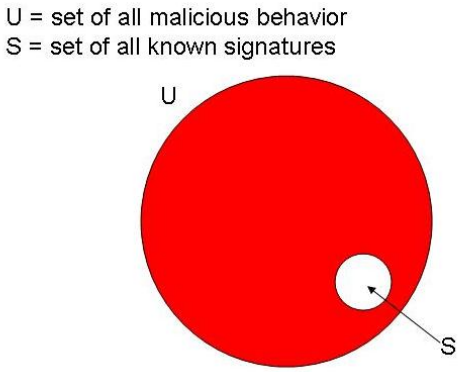

Figure 3: An illustration of why signature-based detection is insufficient.

Some of the examples of signature-based detection techniques attempt to leverage that much of malware are derivatives of previous malware. To take advantage of this observation, malware signatures must be constructed in a manner that captures the malicious essence, or invariant, of the malware being modeled. Obviously, the reason this paradigm of signature construction is used is to make malware detectors less sus-

ceptible to obfuscations. Another salient reason for building signatures in this manner is to minimize the number of malware signatures that are stored in the repository. Although currently, storage is not an issue, over time, storage could potentially become a serious one as this will have direct affects on the time complexity of the malware detector.

4.3.1 Dynamic Signature-based Detection

Dynamic signature-based detection is characterized by using solely information gathered during the execution of the PUI to decide its maliciousness. Dynamic signature-based detection looks for patterns of behavior that would reveal the true malicious intent of a program.

Rule based IDS Approach

In Ilgun et al.'s work, [22] attacks are modeled as state transition diagrams. Assuming that there is some mechanism for auditing data, this mechanism passes this data to a preprocessor which formats the data in a manner that can be analyzed with a state transition diagram. This data is then compared against known penetrations which are in the form of state transition diagrams.

As stated by the authors, this method is susceptible to individuals who gain access to the system by using valid information. For example, an individual who steals a username and password, can use those valid pieces of information to access the system. This approach can only detect identifiable compromises of the system. "Identifiable" are events that are visible changes to the system.

The authors also state that their implementation of this approach, STAT, is generic enough to catch variations of the same attack whereas traditional rule-based tools are less capable in this way, and are therefore lacking in this respect relative to STAT. STAT also has the ability to detect colluding attacks because it maintains a list of the users who have contributed to sequences it captures. Traditional IDSs also lack in this respect.

Behavioral Approach to Worm Detection

Ellis et al. [15] propose a signature-based method for worm detection that is based on known malicious behaviors. The authors present four different behavioral signatures. Base signatures are those which can be identified by monitoring the data flows

coming in and going out of a single node. A base signature is when a server changes into client. Since the worm must propagate itself, after compromising the server, it must once again act like a client to another host in hopes of infecting more machines typically via the same vulnerability. This approach to detection is not as effective if being applied in a peer-to-peer environment.

Another base signature is alpha-in and alpha-out. This base signature simply says that the worms typically send similar data across nodes, and therefore often have similar if not the same ingress and egress data flow links. This approach is limited in that for some services, it is not unusual for them to send out similar data. For example, there is nothing unusual about file servers receiving and sending similar data.

Another form of behavioral signature is called fanout. Fanout simply places a threshold on the number of descendants a host can have at any given time. The descendant relation is an example of an inductive signature. Inductive signatures assume that there is some set of infected hosts responsible for infecting other hosts, which in turn infect more than one other host, causing an exponential growth in infections and a great indicator that a worm is spreading rapidly. In order to create a signature for this type of a behavior, thresholds must be set on the following: (1) the tree's depth, (2) number of descendants in the tree, (3) average branching factor, and (4) the time it takes to reach a particular tree depth.

The authors analyzed the server-to-client signature and the alpha-in/alpha-out signatures. The server to client signature was found to be perfectly sensitive to active worms that changes a server to a client. The alpha-in/alpha-out signature is dependent on the threshold value chosen for alpha. For example, an alpha value of 1, would be very unhelpful as the false alarm rate would be exceedingly high.

4.3.2 Static Signature-based Detection

Static signature-based detection is characterized by examining the program under inspection for sequences of code that would reveal the malicious intent of the program. The goal is to access code which represents the behavior of the program. Static analysis of this code provides an approximation to the run-time behavior of the executable under inspection. Signatures are typically represented by sequences of code. The signature-based method uses its knowledge (e.g. sequences of instructions considered responsible for bringing about malicious behavior) and compares the PUI with the

known signatures for a match. A major advantage of the static signature-based method is that the PUI can be analyzed and maliciousness accurately determined without having to run the executable.

SAVE

Sung et al. [47] propose a method called Static Analysis for Vicious Executables (SAVE). The form of the signature for a given virus is given by a sequence of Windows API calls. Each API call is represented by a 32-bit number. The most significant 16 bits correspond to the module the API call belongs to, whereas the least significant 16 bits corresponds to the API functions position in a vector of API functions. The Euclidean distance is calculated between the known signatures and the sequence of API calls found in the program under inspection. The average of three similarity functions gives the similarity of the PUI's API sequence with that of the signatures from the repository. If the difference is 10 percent or less, then the PUI is flagged as malicious.

Sung et al. compared SAVE to 8 malware detectors. The malware detectors SAVE was compared to were Norton, McAfee Unix Scanner, McAfee, Dr. Web, Panda, Kasperksy, F-Secure, and Anti Ghostbusters. Sung et al. tested all these scanners against variants of W32.Mydoom, W32.Bika, W32.Beagle, and W32.Blaster.Worm. SAVE was the only detector that was able to detect all variants of the aforementioned malware in the study.

Semantics-aware

In the work of Christodorescu et al. [11] malware signatures are represented by templates. Each template is a 3-tuple of instructions, variables, and symbolic constants. Templates attempt to generalize the signature of a malware instance and yet maintain the essence of the malicious code's behavior.

Three steps are needed to identify whether or not the PUI is malicious. First, the PUI is converted into a platform independent intermediate representation (IR) which is a variant of the x86 language. Next, a control flow graph is computed for the intermediate representation of the PUI, and is compared to that control flow graph of the template. Finally, comparison is done via the use of def-use pairs. If for each def-use pair found in the template, there is a corresponding def-use pair in the IR of the PUI, then the program is malicious.

Christodorescu et al.'s results indicate their template based approach has the ability

to detect malware variants with zero false positives. The 21 email worm instances were derived from the following malware families: Netsky, B[e]agle, and Sober. The impressive detection ability was achieved with only 2 templates, namely a decryption loop and a mass mailing template. The mass mailing behavior of the Sober worm was not detected; however, this was because the implementation developed at the time of the work did not support the Microsoft runtime library.

Christodorescu et al. also found that on 2,000 benign Windows programs, their algorithm did not produce any false positives. The authors also evaluated their methods resilience to obfuscation, namely garbage insertion. They used 3 types of garbage insertion methods. One method is the nop insertion, which is the insertion of nop instructions. Another garbage insertion method is stack-op insertion, which inserts stack operations that do not change the semantics of the malware. Lastly, math-op insertion is used, where arithmetic operations are inserted into the malware. B[e]agle.Y was chosen as the malware to obfuscate. Christodorescu et al.'s algorithm outperformed McAfee virus scan for nop insertion, stack-op insertion, and math-op insertion by 25 percent, 75 percent, and 90 percent respectively.

Generic Virus Scanner

Kumar and Spafford [27] proposed a general scanner which detected viruses based on regular expression matching. At each nibble in an input stream (e.g. a file) being scanned, the pattern matching algorithm compares all known viruses matching this nibble value to see if the input stream sequence matches a known virus. Because the scanner proposed was made exclusively for SunOS, the authors' implementation was, consequently, not easily comparable to other virus scanners of different Operating Systems and file systems.

Static Analyzer for Executables (SAFE)

Christodorescu and Jha [9] proposed SAFE that has the ability to take patterns of malicious behavior and create an automaton from it. This automaton has uninterpreted symbols in it that can later be bound to elements present in the executable being inspected. For example, an uninterpreted symbol may be bound to a register name. This register name would come from the CFG that is created from the PUI during verification. If any of the automaton modeling malicious code is found in the CFG then the executable is considered malicious. In the experiments conducted by the authors,

SAFE had a false positive and a false negative rate of zero.

Honeycomb

Kreibich and Crowcroft [26] proposed honeycomb, which is a system that uses honeypots to generate signatures and detect malware stemming from network traffic. The authors' technique operates under the assumption that traffic which is directed to a honeypot is suspicious.

Honeycomb stores the information regarding each connection, even after the connection has been terminated. The number of connections it can save is limited. The reassembled stream of the connection is stored. The Longest Common Subsequence (LCS) algorithm is used to determine whether a match is found between connections stored, and new connections that honeycomb receives.

Since honeycomb needs a set of signatures to compare to, initially it uses anomalies in the connection stream to create signatures. For example, if odd TCP flags are found, then a signature will be generated for that stream. The signature is the stream that came into honeycomb modulo honeycomb's responses to the incoming stream.

To detect malicious code, horizontal and vertical detection schemes are used. In the horizontal approach, the last (nth) message of an incoming stream is compared to the nth message of all streams stored by honeycomb. In the vertical approach, the messages are aggregated, and the LCS algorithm is run on the aggregation of the newly arrived stream as well as the aggregated form of the streams stored by honeycomb.

Signatures that are not used much are deleted from the queue of signatures. Also if a new signature is found to be the same, or a subset of an already existing signature, it is not added to the signature pool. This helps keep the signature pool as small as possible. At regular intervals signatures found are reported and logged to another module. In Kreibich and Crowcroft's empirical study, they were able to develop "precise" signatures for the Slammer and CodeRed II worms.

MEDiC

NMTMEDiC stands for New Mexico Tech's Malware Examiner using Disassembled Code [46]. The method proposed by Sulaiman et al. attempts to identify malicious code by comparing assembly code of the PUI with known malicious signatures. The PUI is disassembled using the PE Explorer producing ASM code. Since assembly code is

essentially a collection of key/value pairs where the key is the code label, and the value corresponds to the instructions for the given label, MEDiC compares programs on the basis of these key/value pairs. A dictionary threshold is used to determine whether or not a key/value pair is "important." The key/value pairs deemed important are recorded as checkpoints. The virus threshold decides whether a PUI is malicious. The virus threshold is the lowest ratio of matches over the number of checks performed. MEDiC has three scanning phases. In the first phase, the key/value pairs are compared with those in the signature set. If the virus threshold is not surpassed, then MEDiC proceeds to the second phase. In the second phase, comparisons are only done for the value component of key/value pair effectively disregarding the key component. This phase is robust against malicious code that changes the name of program labels in an attempt to obfuscate its attack. The third phase is depicted as a "more thorough" process. There is a search threshold which relaxes the matching constraint. The search threshold allows for some minimum number of instructions to match before MEDiC decides that it has found a match between a known signature's instruction and the PUI's instruction.

When compared to 13 different viruses, MEDiC outperformed Norton, McAfee, Dr. Web, Panda, Kaspersky, F-Secure, and Pccilin. The 13 viruses were variants of the Dos, CodeGreen, Aircop, Badboy, and Yaha. Also to the credit of MEDiC, no false positives were observed.

4.3.3 Hybrid Signature-based Detection

The hybrid signature-based detection approach uses static and dynamic properties to determine the maliciousness of the PUI.

Analyzing and Detecting Malicious Mobile Code

Mori et al. [37] propose a tool that detects mobile self-encrypting and polymorphic viruses. Traditional pattern matching techniques cannot detect these types of viruses as they are designed to circumvent pattern matching techniques. In other words, by encrypting itself, viruses do not exhibit the patterns which pattern matching techniques rely on. Hence the authors created a technique and tool to address this issue.

Using Mori et al.'s technique, the code is allowed to decrypt itself in an emulator of the Operating System. Once the virus has decrypted its payload, static analysis is performed on it to identify system calls made from the payload. Detection policies are

modeled as state machines which represent malicious behavior. The user specifies the detection policy (or signature). When a match occurs, the mobile application is considered malicious. In an experiment, their tool was able to detect 600 virus/worm samples.

Worm vs. Worm

Castaneda et al. [5] propose a method which uses the honeypot IDS for capturing malicious processes. The honeypot IDS uses a signature-based method to capture the malicious process. Once the malicious process has been captured, the algorithm attempts to find the malicious payload. The algorithm does this by making a copy of the executable and starting at some address and overwriting it with an anti-worm payload. This modified executable is sent to a virtual machine that runs the executable, and if it crashes or exhibits some sort of unexpected behavior, then the start of the malicious payload was not correctly identified. So the next step of the algorithm involves incrementing from this beginning address until the malicious payload is found, which is indicated by a successful run of the executable. This process is the function of the anti-worm–finding the malicious payload and overwriting it.

Castaneda et al. evaluated four different anti-worm propagation schemes via simulation. The worm used in the authors' simulation was the Code-Red I version 2 (CRIv2). Each simulation started with 360,000 vulnerable hosts.

In the passive anti-worm scheme, the anti-worm listens for connections from an infected host. When the anti-worm detects packets from an infected machine the counter-attack is executed. The number of cured hosts on the Internet initially determines the effectiveness of the passive anti-worm scheme. The number of initially cured hosts needs to be closer to 10 percent of the hosts online in order to control outbreaks.

In the active anti-worm scheme, the anti-worm has just as many threads as its malicious counterpart (100 threads). The anti-worm performs random scanning of hosts. This approach was effective in keeping infected hosts below 15 percent of the total hosts on the Internet in Castaneda et al.'s simulation. This approach does have some important negative aspects. For example, this approach creates more network traffic and could cause unnecessary bottlenecks and other networking issues.

In the active-passive hybrid anti-worm scheme, the anti-worm actively seeks infected hosts until some condition is met. When the condition is met, the anti-worm

enters the passive anti-worm scheme. An example of a condition could be timer. For instance, after three hours of the active anti-worm scheme, the anti-worm changes into the passive scheme. The hybrid was just as successful as the anti-worm scheme, with less network traffic.

The IDS based anti-worm scheme was the last propagation scheme evaluated by the authors. The idea is that there would be IDS sensors deployed throughout the Internet. These sensors would detect when malicious traffic was traveling between nodes. Once malicious packets were identified, anti-worms will be sent to the sender and intended recipients. The effectiveness of this approach hinges on the packet capture rate of the sensors that would be placed at various nodes around the Internet. For instance, with a packet capture rate of .2 percent, the number of infected hosts is well below 15 percent. With a lower packet capture rate of .05 percent, the number of infected hosts appears to be over 30 percent of the total hosts.

MCF: Malicious Code Filter

Lo et al. [33] propose a tool, Malicious Code Filter (MCF). This is a tool for analyzing executables. It does not require the programmer to provide formal specifications. The method takes an executable, and creates an intermediate representation which is a CFG of the executable. Through an analysis of various malware, the authors developed "tell-tale" signs for malware. In this technique, the human analyst ultimately makes the decision about whether some program is malicious. The tell-tale signs serve as guidelines for the human analyst. Tell-tale signs are unique pervasive properties malware tend to exhibit. These tell-tale signs describe various classes of malware. Henceforth, a programmer would create a filter that would be based on some tell-tale sign. For example, `fingerd` has a vulnerability where an intruder can gain access to a protected file by linking the `.plan` file after the system checks that `.plan` is not a symbolic-link and before the `open()` system call is executed. This would be detected by the "Race Condition" sign.

There are three classes of tell-tale signs. One class consists of those signs which are identified using program slicing. The Race Condition tell-tale sign would be an example of a sign that uses program slicing to identify vulnerabilities. Another class consists of those signs that is based on some form data analysis, that does not necessitate the use of program slicing. An example of a relevant data analysis would be pointer analysis. "Well-Behavedness" is an example of tell-tale sign of this class. This

tell-tale sign mandates that checks be done on dereferenced pointers to ensure they have valid addresses and that pointers and arrays do not overflow. The third class of tell-tale signs requires human expertise and may utilize program slicing. A tell-tale sign from this class is "Identification of Changes." For example, if a user has what he writes maliciously altered before it reaches the reading application on his system, slicing for the `write()` and `read()` system calls may be useful. An example of how the technique works on a .c file is given in the paper; however, no empirical study evaluating the method is given.

Malware Pattern Scheme

Using combinatorial design, Filiol proposes a novel scheme for detecting malware in [16]. The goal of Filiol's work is to provide a malware pattern (signature) detection scheme that is robust against black-box signature extraction techniques. The idea is that the signature can be divided into a number of sub-signatures that can reliably detect malware independently. If implemented in virus scanners, each virus scanner would use a particular sub-signature to detect malware represented by the signature. The actual sub-signature used would be determined by factors such as the serial number of the processor, the MAC address, the owner's user name and email address, and some password. Each owner of a virus scanner would have a fixed sub-signature that is used for each signature in the virus scanner's repository. This increases the difficulty of a malware writer to successfully create a varied malware based on his or her black-box analysis. For example, if a malware writer is able to extract the sub-signature necessary to circumvent a virus scanner on one user's machine using malware *M'*, there is no guarantee that *M'* will be able to circumvent virus scanners on different machines because chances are they will be using different sub-signatures to detect to malware. The scheme proposed by Filiol would effectively decrease the rate at which malware would spread since finding a way to circumvent one virus scanner (that is based on black-box signature extraction) would not necessarily suggest that every virus scanner of the same model would be susceptible to the modified malware.

The proposed scheme has not been empirically tested. The author is working on the implementation of his proposed scheme to explore his scheme's feasibility.

5 Summary

In this survey we have presented a series of techniques, examples, issues, and topics within the area of malware detection. We have proposed a novel classification scheme for malware detection techniques. We have also identified inadequacies in the signature-based and anomaly-based (specification-based) detection methods.

Table 1 classifies the detection techniques described in this survey. The sample of malware detection techniques depicted here is an indicator of current trends in malware detection and raises some interesting questions. For example, why are there so few (only one in this survey) static anomaly-based–that are not specification-based–techniques ? Is specification-based detection the most promising malware detection technique–as it seems to have the most techniques in the literature ? Does the majority of the research community simply find anomaly-based and specification-based detection more interesting than signature-based detection ? Given Figure 2 and Figure 3, should we be contemplating other strategies for malware detection ?

To answer the raised questions effectively, a universal metric for malware detection ability needs to be developed. Since certain Operating Systems and file systems are more susceptible to malware attacks than others, this should also be considered in the development of a metric system. Currently, evaluation of malware detection techniques seem rather arbitrary. When the creator of technique A compares his technique to technique B, it is usually unclear whether technique B has its parameters optimized for detection. It is also unclear whether the set of test subjects (malware) used for techniques A and B had properties that made technique A appear better, and if this is case, it is unclear whether there is possibly another set of test subjects that would make technique B appear better.

The literature suggests that COTS malware detectors are easily obfuscated. Every malware detector prototype from the literature that is compared to COTS malware detectors outperforms them. Given the current state of malware detection research, comparison to COTS malware detectors should be treated as a sign of baseline achievement and not a sign of a particularly strong malware detection technique.

Table 1: A summary of publications that introduce malware detection techniques.

Malware Detection Method	Approach	Examples
Anomaly-based	Dynamic	[4], [21], [30], [40], [42], [44], [48], [51], [54]
	Static	[31]
	Hybrid	[17] , [52]
Specification-based	Dynamic	[23], [24], [25], [29], [32], [34], [36], [41], [43], [45], [55]
	Static	[1], [2], [3], [14]
	Hybrid	[13], [18], [20], [38], [49], [50]
Signature-based	Dynamic	[15], [22]
	Static	[9], [11], [26], [27], [46], [47]
	Hybrid	[5], [16], [33], [37]

References

[1] F. Adelstein, M. Stillerman, and D. Kozen. Malicious code detection for open firmware. *In Proceedings of the 18th Annual Computer Security Applications Conference*, 2002.

[2] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*, 2001.

[3] J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behavior. *In 8th Workshop on Enabling Technologies: Infrastructure for Collaborative Entrerprises*, 1999.

[4] M. Boldt and B. Carlsson. Analysing privacy-invasive software using computer forensic methods. *http://www.e-evidence.info/b.html*, January 2006.

[5] F. Castaneda, E. C. Sezer, and J. Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. *Proceedings of the 2004 ACM Workshop on Rapid Malcode*, 2004.

[6] CERT/CC, Carnegie Mellon University. *http://www.cert.org/present/cert-overview-trends/module-2.pdf*, May 2003.

[7] CERT/CC, Carnegie Mellon University. *http://www.cert.org/present/cert-overview-trends/module-4.pdf*, May 2003.

[8] CERT/CC, Carnegie Mellon University. *http://www.cert.org/stats/cert_stats.html\#incidents*, last updated: April 2006.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. *Usenix Security Symposium*, 2003.

[10] M. Christodorescu and S. Jha. Testing malware detectors. *In Proceedings of the International Symposium on Software Testing and Analysis*, July 2004.

[11] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. *In Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[12] M. Ciubotariu. Netsky: a conflict starter? *Virus Bulletin*, May 2004.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *In Proceedings of the 7th USENIX Security Conference*, Jan. 1998.

[14] M. Debbabi, E. Giasson, B. Ktari, F. Michaud, and N. Tawbi. Secure self-certified cots. *In Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 183–188, 2000.

[15] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. A behavioral approach to worm detection. *In Proceedings of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, 2004.

[16] E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal of Computer Virol.*, 2006.

[17] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination. *In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, May 1994.

[18] J. T. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. *11th USENIX Security Symposium*, 2002.

[19] J. Gordon. Lessons from virus developers: The beagle worm history through april 24, 2004. *SecurityFocus*, May 2004.

[20] W. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. *In Proceedings of the 20th IEEE/ACM Internation Conference on Automated Software Engineering*, pages 174 – 183, 2005.

[21] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, pages 151 – 180, 1998.

[22] K. Ilgun, R. A. Kemmerer, and Porras P. A. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 1995.

[23] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. *In the 21st ACM Symposium on Applied Computing (SAC)*, 2006.

[24] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. *In Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 1994.

[25] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[26] C. Kreibich and J. Crowcroft. Honeycomb – creating intrustion detection signatures using honeypots. *In 2nd Workshop on Hot Topics in Network*, 2003.

[27] S. Kumar and Spafford E. H. A generic virus scanner in c++. *In Proceedings of the 8th Computer Security Applications Conference*, pages 210 – 219, 1992.

[28] C. Landwehr, A. Bull, J. McDermott, and W. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[29] R. B. Lee, D. K. Karig, P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. *International Conference on Security in Pervasive Computing (SPC)*, 2003.

[30] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. *In Proceedings of the 7th USENIX Security Symposium*, 1998.

[31] W. Li, K. Wang, S. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. *6th IEEE Information Assurance Workshop*, June 2005.

[32] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. *Usenix Security Symposium*, 2005.

[33] R.W. Lo, K.N. Levitt, and R.A. Olsson. Mcf: Malicious code filter. *Computers and Society*, pages 541–566, 1995.

[34] W. Masri and A. Podgurski. Using dynamic information flow analysis to detect attacks against applications. *In Proceedings of the 2005 Workshop on Software Engineering for secure sytems –Building Trustworthy Applications*, 30, May 2005.

[35] G. McGraw and G. Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–44, 2000.

[36] M. Milenkovic, A. Milenkovic, and E. Jovanov. Using instruction block signatures to counter code injection attacks. *ACM SIGARCH Computer Architecture News*, 33:108–117, March 2005.

[37] A. Mori, T. Izumida, T. Sawada, and T. Inoue. A tool for analyzing and detecting malicious mobile code. *In Proceedings of the 28th International Conference on Software Engineering*, pages 831 – 834, 2006.

[38] J. Rabek, R. Khazan, S. Lewandowski, and R. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. *In Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pages 76–82, 2003.

[39] San Diego Supercomputer Center. *http://security.sdsc.edu/incidents/worm.2000.01.18.shtml*, June 2002.

[40] I. Sato, Y. Okazaki, and S. Goto. An improved intrusion detection method based on process profiling. *IPSJ Journal*, 43:3316 – 3326, 2002.

[41] S. E. Schechter, J. Jung, and Berger A. W. Fast detection of scanning worms infections. *In Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection (RAID) 2004*, 2004.

[42] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anomalous program behaviors. *In IEEE Symposium on Security and Privacy*, 2001.

[43] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. *USENIX Intrusion Detection Workshop, 1999.*, 1999.

[44] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: A new approach for detectin network intrusions. *ACM Computer and Communication Security Conference*, 2002.

[45] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. *International Conference Architectural Support for Programming Languages and Operating Systems*, 2004.

[46] A. Sulaiman, K. Ramamoorthy, S. Mukkamala, and A.H. Sung. Malware examiner using disassembled code (medic). *Systems, Man and Cybernetics (SMC) Information Assurance Workshop 2005*, June 2005.

[47] A. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables (save). *In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, 00:326–334, 2004.

[48] C. Taylor and J. Alves-Foss. Nate – network analysis of anomalous traffic events, a low-cost approach. *New Security Paradigms Workshop*, 2001.

[49] A. Vasudevan and R. Yerraballi. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. *In Proceedings of the 29th Australasian Computer Science Conference*, pages 311–320, 2006.

[50] D. Wagner and D. Dean. Intrusion detection via static analysis. *IEEE Symposium on Security and Privacy*, 2001.

[51] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. *In Proceedings of the 7th International Symposium on (RAID)*, pages 201–222, September 2004.

[52] Y. M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. *In Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 368–377, 2005.

[53] N. Weaver, V. Paxon, S. Staniford, and R. Cunningham. A taxonomy of computer worms. *In Proceedings of the 2003 ACM Workship on Rapid Malcode*, pages 11–18, 2003.

[54] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. *Recent Advances in Intrusion Detection (RAID)*, 2000.

[55] J. Xiong. Act: Attachment chain tracing scheme for email virus detection and control. *In Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, 2004.