AGIS: Towards Automatic Generation of Infection Signatures

Zhuowei Li[†], XiaoFeng Wang[†], Zhenkai Liang[§] and Michael K. Reiter[£]
[†]Indiana University at Bloomington.[§]Carnegie Mellon University.[£]University of North Carolina at Chapel Hill.

Abstract

An important yet largely uncharted problem in malware defense is how to automate generation of infection signatures *for detecting compromised systems, i.e., signatures that characterize the* behavior *of malware residing on a system. To this end, we develop AGIS, the first host-based technique that detects infections by novel malware and automatically generates an infection signature of the malware. AGIS monitors the runtime behavior of suspicious code according to a set of security policies to detect a previously undetected infection, and then identifies its characteristic behavior in terms of system or API calls. AGIS then statically analyzes the corresponding executables to extract the instructions important to the infection's mission. These instructions can be used to build a template for a static-analysis-based scanner, or a regular-expression signature for legacy scanners. AGIS also detects encrypted malware and generates a signature from its plaintext decryption loop. We implemented AGIS on Windows XP and evaluated it against real-life malware, including keyloggers, mass-mailing worms, and a well-known mutation engine. The experimental results demonstrate the effectiveness of our technique in detecting new infections and generating high-quality signatures.*

1 Introduction

The capability of malware to spread rapidly has motivated research in fully automated defense techniques that do not require human intervention. For example, significant strides have been made in the automated generation of *exploit signatures* and patches (e.g., [40, 27, 24, 34, 32, 44, 33, 31, 12, 14, 47, 30, 39]) to protect vulnerable software from being exploited. These approaches detect the compromise of a process and then trace the compromise to the exploit input that caused it, enabling the construction of a signature for that input and possibly variations thereof. These techniques, however, are largely constrained to detecting and generating signatures for code-injection attacks, due to the limited class of violations they can detect.

Although many research projects have developed solutions to automatically generate exploit signatures to *prevent* the malware from penetrating into vulnerable systems, they cannot prevent all attacks, especially zero-day ones, and thus allow malware to infect the victim systems. This problem calls for an automatic mechanism to *detect* the malware when it has already penetrated into the vulnerable systems. We meet this challenge by exploring the automatic generation of a different type of signature, named as *infection signature*, which characterizes malware's behaviors when they reside on a system. The main objective of constructing infection signatures is to detect the presence of a malware that has successfully penetrated into a system. While an exploit signature can be generated through analyzing the software vulnerability which allows the exploit to happen [47, 8], infection signatures are generally more difficult to get, due to the diversity of malware's behavior in an already infected system.

The first kind of infection signatures to have undergone extensive study are virus signatures, which are generated mostly through manual analyses of virus code. Kephart and Arnold proposed an approach that automatically extracts invariant byte sequences from "goat" files infected by the virus running in a controlled environment [22]. A similar approach has been adopted by Symantec in their digital immune system [43]. These techniques rely on a virus' replication behavior, which is absent in other types of malware such as spyware, Trojans and back doors. In addition, they cannot handle polymorphic and metamorphic code [9].

There are other malware detectors that identify malware code using very simple techniques like the MD5 checksum. Generation of a checksum signature can be easily automated. However, it is too specific to accommodate any modification to the code such as injection of NOP instructions. Wang et al. [45] proposed a network-based signature generation approach which automatically extracts invariant tokens from malware's communication traffic. However, such a signature can be evaded if attackers vary the servers which communicate with infected hosts (possibly through a botnet) or simply encrypt network traffic.

In this paper, we seek a very general approach to automatically generating infection signatures, in particular one that does not presuppose a method by which the attacker causes his code to be executed on the computer; in the limit, the user could have installed and run the malware himself, as users are often tricked into doing so. Consequently, our approach does not begin with detectors for a code-injection attack (e.g., using an input-provided value as a pointer [28]), but rather monitors for an array of suspicious behaviors

that are indicative of a compromise, such as a system call to hook a dynamic-link library (DLL) file for intercepting keystrokes and subsequent I/O activities for depositing and transferring a log file. Once such behavior are detected, our technique employs dynamic and static analyses to extract the instruction sequences used to perform the offending actions, and can do so even if the instructions have undergone moderate obfuscations. These instructions can be used to build a "vanilla" version of infection [10, 11], an instruction template for a static analyzer to detect the infection's variants, or regular-expression signatures for legacy malware scanners. In the case that malware has been encrypted, our technique extracts the instructions necessary for it to decrypt its executable and run, which must be plaintext. We have implemented these techniques in a system called AGIS, and will detail its operation here.

At a high level, AGIS bears some similarity to recent work on behavior-based spyware detection that composes dynamic and static analysis to detect spyware in the form of a plug-in to Internet Explorer [26, 16]. However, our technique complements that approach in that it works on standalone malware such as keyloggers, mass-mailing worms. Most recently, Yin et al. [48] proposed Panorama to utilize instruction-level taint propagation for malware detection and analysis. In this aspect, Panorama could act as the first part of AGIS for infection detection. Our experiments in AGIS, nevertheless, shown that our lightweight, coarser-grained taint propagation in the system-call level was enough for infection detection **by successfully generating signatures for all infections**. Moreover, it practically tackles the problems in Panorama, such as indirect dependencies, anti-emulation techniques, and a high performance penalty. Lastly but most importantly, AGIS is the first host-based system which automates *infection* signature generation for a variety of infections.

We believe that AGIS advances research on malware defense in the following respects.

• **Detection of infections caused by novel malware**. We have developed a new technique to detect a previously unknown infection by monitoring behavior of suspicious code for violations of security policy. Examples of such behavior include hooking a DLL file and exporting log files, or recursively searching a file system (for email addresses) and connecting to SMTP servers. While our technique is also applicable to plugin-based spyware like in [26], our current focus is given to standalone malware.

• **Automatic generation of infection signatures**. We have developed novel dynamic and static analysis techniques to generate infection signatures. Our dynamic analyzer inputs to the static analyzer the locations of the system or API calls within an infection's executables which are responsible for its malicious behavior, and other information that facilitates static analysis of the malicious code. The static analyzer then extracts the instructions indispensable to these calls. Our approach also keeps track of the relationships among different components of an infection through monitoring their interactions, which enables automatic generation of a series of signatures to identify the infection components which are indirectly responsible for the malicious behavior. This property is particularly important to malware disinfection, as some infection component, if left undetected, could restore other components once removed.

• **Resilience to obfuscated and encrypted infection executables**. We demonstrate that our technique can reliably and efficiently extract signatures from an infection even if its code has been moderately obfuscated and encrypted.

2 Design

To generate infection signatures, AGIS takes two key steps: *malicious behavior detection* and *infection signature extraction* (Figure 1). **In the first step, a malware is penetrated into a sandboxed environment such as honeypot [41]. Followingly, the suspicious executables of the malware are tainted and their runtime activities are monitored and checked against security policies to detect malicious behaviors.** Any detection triggers the static analysis in the second step to extract the instruction sequences responsible for these behavior, from which infection signatures are constructed. In this section, we first describe the general idea through a simple example, and then elaborate on the techniques involved.

2.1 Overview

As an illustrative example, let us consider a Trojan downloader trapped within a honeypot. Once activated, the Trojan downloads and installs a keylogger, and sets a `Run` registry key to point to it in order to survive the infected system reboots. The keylogger consists of two components, an executable file which installs a hook to Windows message-handling mechanism, and a DLL file containing the hook callback function to create and transfer log files.

To detect this infection, the AGIS-enhanced honeypot first runs the Trojan to monitor its system calls which reflect the behavior of the code. From these calls, AGIS constructs an *infection graph* which records the relations among the Trojan and the two files it downloads, e.g., the registry change to automatically invoke the keylogger executable, and extends the surveillance to them. An alarm is raised when the keylogger installs the DLL to monitor keyboard inputs through the system call `NtUserSetWindowsHookEx`, and the DLL exports a file in response to inputs of keystrokes.[1] Such behavior is suspected to violate a security policy which forbids hooking

[1] Keystrokes are automatically generated by a program in AGIS.

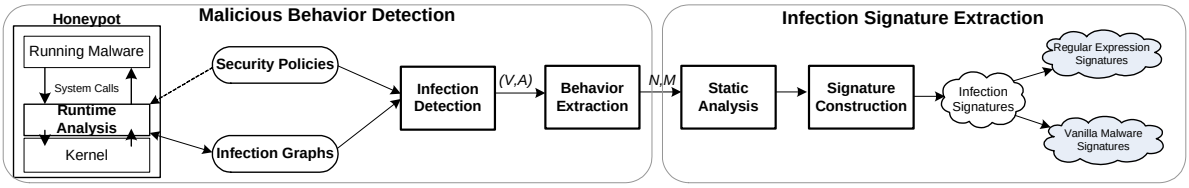

Figure 1. The design of AGIS.

the keyboard and writing a log file. The presence of this malicious activity can be confirmed by a static analyzer which tries to find an execution path from the callback function in the DLL to the `NtWriteFile` call being observed. Backtracking on the infection graph, AGIS also pronounces the Trojan to be malicious.

To extract infection signatures, our dynamic analyzer first identifies the locations of the calls within executables (a.k.a. *call sites*) responsible for the malicious behaviors which include downloading of the keylogger, modification of the registry key, invocation of the keylogger, installation of the DLL and export of a log file. It can also collect other information useful to static analysis, in particular, the call sites of other system calls being observed, anchoring the execution path of the program. Using such information, a static analyzer extracts the instruction sequences in individual executables which affect the malicious calls directly or transitively. The infection signatures of the Trojan downloader are derived from these instructions.

2.2 Malicious Behavior Detection

The objectives of this step are to determine whether a piece of suspicious code is a malware's infection and if so, to identify a set of behaviors which characterize the infection. AGIS adopts a novel technique which first builds an infection graph to describe the relations among different components of an infection such as modified registry keys and downloaded executables, and then detects some components' malicious behaviors using a set of security policies as well as related activities from other components. These behaviors are used to generate infection signatures.

Infection Graph. An infection graph can be described as a tuple $\langle V, A\rangle$, where V is a set of vertices and A is a set of arcs. Set V is further partitioned into two subsets: a set S of subjects which contains executable components such as a keylogger and a set O of objects that includes other components such as registry entries. An arc a from component v to v' indicates that either v outputs something to v', e.g., creating v', or v' inputs something from v, e.g., reading from v. We also consider an arc existing from an auto-start extensibility point (ASEP) [46] such as the Run registry key to the executable it points to.

Our approach identifies an infection graph using a system call level taint-analysis technique. We first taint the suspicious code trapped in a honeypot and its process, which

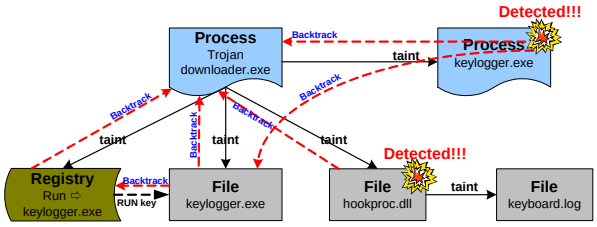

Figure 2. The infection graph of the example. The dotted lines annotated with 'backtrack' describe the backtracking process. The vertices with 'Detected!!!' are detected violating security policies.

forms the first set of vertices on the infection graph we called *the sources*. Other vertices are obtained through taint propagation: a tainted v propagates taint to another subject or object v' if an arc can be drawn from v to v' as discussed above. Figure 2 presents the infection graph of the example in Section 2.1, in which the Trojan passes the taint to the Run registry key, the hook installer, and the DLL file.

Security Policies. Tainted executables are monitored by AGIS for the behaviors that violate a set of predetermined security policies. Infections of the same type usually exhibit common behavior patterns. For example, a keylogger usually hooks the system message-handling mechanism and then records keystrokes into a local or remote log; a mass-mailing worm is very likely to search the file system for email addresses and then connect to remote SMTP servers to propagate itself to other clients. Security policies are set to flag an alarm whenever these malicious activities are observed. For the above example, the keylogger policy is used to forbid the behavior sequences of hooking and recording, and the mass-mailing policy to prevent the behaviors of reading files and then connecting to SMTP servers. In AGIS, we specify the security policy using Behavior Monitoring Specification Language (BMSL) [38].[2] Table 1 describes the two example policies in BMSL.

A security policy can capture a large number of malware instances: for example, we examined 23 mass-mailing worms reported by Symantec [1], all of which exhibit the behaviors described above. Our experiments on 19 common applications in Section 4.1 also shown that the above example polices do not introduce any false positives. AGIS can incorporate more such security policies for infection detection. However, our main objective in this paper is to propose a new automatic generation mechanism of infection

[2] BMSL is an event-based language designed for policy specifications. BMSL rules have the form $event\_pattern \rightarrow action$, where both $event\_pattern$ and $action$ can be defined as regular expressions to connect functions and statements.

signatures, and as such we use the two security policies for keyloggers and mass-mailing worms, which are described above, in the following sections. Exploring the design of security polices is left as our future work.

Infection Detection and Behavior Extraction. AGIS detects an infection by matching the behaviors of suspicious code to the event pattern on a security policy. Most of such behaviors can be directly observed through system calls, while the rest needs to be identified through static analysis of suspicious executables. For example, the keylogger rule in Table 1 will be activated only if the program makes $WriteFile$ or $Sendto$ calls and those calls are reachable from the hooked function f. The second condition is verified by the helper function $ExistPath$, which searches an execution path connecting the callback function (pointed by the hook call) to a function exporting a file on the control flow graph (CFG) of a tainted executable[3]. The $ExistSearchLoop$ helper function in the mass-mailing rule can be implemented using dynamic analysis alone: our approach triggers the rule if the frequency of recurrence of $ReadFile$ or related calls from the same call site exceeds a pre-determined threshold.

Once an event pattern is observed, AGIS announces detection of an infection and puts the detected processes and their executable files to the infection set $\mathcal{N}$. After that, the $backtrack$ function is invoked, which continues to add into $\mathcal{N}$ the vertices on the infection graph with arcs to the vertices inside $\mathcal{N}$ (called responsible arcs) until all such vertices are included in that set. During this process, the file of every vertex in $\mathcal{N}$, which could also be a vertex, is also included in the infection set. These vertices and their responsible arcs form a subgraph connecting the sources to the behaviors that trigger a security policy. We further remove the vertices which do not have physical representations on the hard disk and their arcs. The remaining subgraph records all the behaviors both necessary for the malicious activities to occur and retrievable from a compromised system. We call the set of such behavior the *infection action set*, denoted by $\mathcal{M}$, which is used to generate signatures for every file in $\mathcal{N}$.

Figure 2 also illustrates a detection and backtrack process. Here the malicious behaviors include the actions to hook and record keystrokes from the keylogger, the actions to change the run registry key and deposit the keylogger from the installer, and the registry entry that points to the keylogger which automatically starts the malware.

2.3 Infection Signature Extraction

An ideal infection signature should uniquely characterize an infection to eliminate false positives, and also tolerate metamorphism exhibited by malware variants to avoid false negatives. AGIS pursues these two goals though extracting the instruction sequences responsible for an infection's behaviors in its infection action set $\mathcal{M}$. These behaviors are important in a sense that they are indispensable to malware's mission. Therefore, their corresponding instruction sequences could offer a unique characterization of the infection caused by the malware.

Our approach utilizes a composition of dynamic analysis and static analysis to extract the important instruction sequences. This approach works well against the code with moderate obfuscation, as we discovered in our study.

Dynamic Analysis. An executable's behaviors observed by AGIS are in the form of system calls. Our dynamic analyzer intercepts these calls and examines their call stacks to find out the return addresses insides a tainted executable's process image. These addresses are further mapped into the call sites in the executable's physical file. This approach is able to work smoothly for the programs that do not contain any encoded components. For an encoded executable, the approach reveals the discrepancy between the instructions in its process image and those in its file, which allows the static analyzer to extract the code indispensable for decrypting and running it.

A problem here is that a call's stack frame can be forged by the malware. For example, an internal callee can first wipe out any stack frame and recover it before returning to its caller. However, the stack frame is difficult to fake if the callee is an API function of which the attacker has little control, and any inappropriate manipulation on the stack will cause crash. Therefore, AGIS only trusts the call sites of API calls, not those of internal functions.

Static Analysis. After locating all the call sites, our static analyzer applies a chopping technique [36][4] to extract the instruction sequences influencing the calls responsible for the malicious behaviors in the infection action set $\mathcal{M}$. Chopping is a static analysis technique which reveals the instructions involved in a transitive dependency from one specific instruction (the source criterion) to another (the target criterion) [36]. For example, to find a chop for a target instruction `call eax`, we first find from the program's control flow the last instruction before the target which operates on `eax`, and then move on to identify the last instruction which influences that instruction, and so on, until the source instruction is reached. Since the behaviors in which we are interested are system or API calls, the objective of the chopping is to find all the instructions which directly or transitively affect these calls. To this end, not only do we need to take the call instruction itself as the target criterion, but we also have to include in the target other instructions

[3]Static analysis can be defeated by anti-disassembling techniques [20], or deep obfuscations of the executable. When this happens, we can use instruction-level dynamic analysis to verify the existence of an execution path.

[4]The chopping algorithm we used is described in Section 3.2. Our approach differs from the standard chopping techniques in that we only identify the related instructions on a particular execution path, instead of all possible paths. Such a path is located using the call sequence observed.

NAME	SECURITY POLICY	COMMENTS
Keylogger rule	$any()*; hook(keyboard, f) \mid f1 = f; any()*; (WriteFile \mid\mid Sendto) \mid ExistPath(f1) \longrightarrow detected(\mathcal{N}) \&\& backtrack(\mathcal{N}) \&\& GenSign(\mathcal{N})$	If a call to hook keyboard is observed in the system call set $SysCall$ and the callback function f it points to has an execution path leading to either $WriteFile$ or $Sendto$, then a keylogger is detected ($detected$) and its processes and files are added into $\mathcal{N}$, the infection set. We also need to backtrack the infection graph, adding the tainted subject or object with an arc to the subject(s) or object(s) in $\mathcal{N}$ to $\mathcal{N}$ ($backtrack$), and generate a signature for every file in $\mathcal{N}$ ($GenSign$).
Mass-mailing-worm rule	$any()*; (ReadFile() \mid ExistSearchLoop; (!Sendto(SMTP))*; Sendto(SMTP) \longrightarrow detected(\mathcal{N}) \&\& backtrack(\mathcal{N}) \&\& GenSign(\mathcal{N})$	If an executable file contains a loop to search directories for reading files ($ExistSearchLoop$) and API calls to send messages to SMTP servers, then it is a mass-mailing worm.

Table 1. Examples of security policies.

known to be part of the call, in particular, the stack operations for transferring parameters. This requires knowledge of an API function's model, which provides the information on the input parameters of the API.

Here we describe the idea behind our static analysis mechanism. Our approach takes advantage of the call sequences observed in the dynamic analysis step to pinpoint the execution path an executable went through in dynamic analysis. If the executable is multi-threaded, we build a call sequence for every thread to make sure that it reflects an execution path. Let $(c_0, \ldots, c_n)$ be the call sites of a call sequence, where c_0 is the beginning of the executable's control flow and c_n is a call site inside the infection action set $\mathcal{M}$ that is determined to violate policy. To extract the instructions responsible for that call, AGIS works as follows: (1) disassemble the executable's binaries and construct a control flow graph; (2) find an executable path p which includes $(c_0, \ldots, c_n)$; (3) for $k = n \ldots 1$: chop the instruction sequence of p between c_{k-1} and c_k.

Metamorphic Infection. AGIS can reliably extract a chop even if an infection has been moderately obfuscated. Common obfuscation transformations [10] include *junk-code injection*, *code transposition*, *register reassignment* and *instruction substitution*. AGIS forms a CFG before chopping which defeats the injection of the junk code unrelated to the malicious calls. The technique proposed in SAFE [10] can also be used to mitigate the threat which adds junk code to a chop. Specifically, code between two points on the chop $[p_1, p_2]$ is deemed as junk code if every variable (register or buffer) has the same value at p_1 as its value at p_2. However, the problem of junk code detection is undecidable in general [10]. The code transposition attack becomes powerless in the presence of the CFG, which restores the original program flow. Register reassignment and instruction substitution are more to do with infection scanning than signature generation, as the objective of AGIS is to identify the instructions indispensable to an infection's mission, not all of their possible variations. A static-analysis based scanner can convert the output of AGIS to an intermediate form [11] which replaces the registers and addresses with variables and utilizes a dictionary to detect equivalent instructions.

Encoded Executables. The dynamic analyzer also compares an infection's instructions around malicious call sites in the virtual memory with their counterparts in the physical file.[5] If there is a discrepancy, AGIS reports that the infection is encoded and moves on to generate a signature from its decryption loop. This is achieved through identifying the instruction which writes to the addresses of these malicious calls, and then chopping the infection's executable to extract all other instructions directly or transitively related to it. These instructions are deemed indispensable for decrypting and running the executable.

A critical question here is how to capture the instruction serving as the chopping target. The most reliable way is using tools such as Microsoft's Nirvana and iDNA [7] to conduct an instruction-level tracing. A more lightweight but less reliable alternative is changing a malicious executable's physical file to set the attribute of the section involving malicious calls to ready-only and rerun the executable. Such a rerun will produce an exception, which reveals the location of the instruction. We evaluated this approach using a real infection (Section 4.2), and successfully extracted the chop for the decryption loop. Its weakness is the possibility to identify a wrong instruction if the read-only section actually contains data.

Construction of Signatures. A collection of the chops from the beginning of the execution flow c_0 to important calls within one thread or process constitutes a piece of vanilla malware, which describes the malicious activities an infection carries out. In case the infection is encrypted, the chop for its decryption loop is treated as vanilla malware.

Compared with a static analyzer, traditional pattern-matching scanners perform much faster though they are much less resilient to metamorphism. AGIS can generate byte-sequence signatures or regular-expression signatures for these scanners. Here is a simple approach. Given a signature size l in bytes, the signature generator selects a malicious call or a decryption instruction and walks from its location backward along its chop to find the first $m + 1$ instruction segments, each of which has a continuous address space and contains B_i $(1 \leq i \leq m + 1)$ bytes. These segments satisfy two conditions: (1) $\sum_{i=1}^{m} B_i \leq l$ and (2) $\sum_{i=1}^{m+1} B_i > l$. A regular-expression signature is formed

[5] A malware could evade this approach by deliberately putting the instructions around call sites to their corresponding locations in the file. This attack can be defeated through extracting the chop of malicious calls from a tainted process's virtual memory and checking its existence in the process's physical file.

through a conjunction of the first m segments and a string in the $(m+1)$th segment with a length of $l - \sum_{i=1}^{m} B_i$ bytes. For example, let the signature size be 30 bytes and the sizes of three segments closest to the call site be 8, 12 and 16; the signature generated is a conjunction of the first and the second segments, and a string of 10 bytes in the last segment. Our research shows that the efficacy of such a signature is related to the selection of the malicious call. If that call has also been frequently used by legitimate programs, a long signature is needed to subdue the false positive rate. Otherwise, a short signature can be sufficient.

3 Implementation

AGIS was prototyped on Windows XP to evaluate its efficacy. Our implementation includes a Windows kernel monitor for dynamic analysis and a static analyzer.

3.1 Kernel Monitor

The monitor hooks Windows system service dispatch table (SSDT) and the shadow table to intercept the system calls from the malware's executable for the purpose of constructing an infection graph and detecting malicious behaviors. It also employs a *stackwalk* technique to identify the call sites of system or library calls. Table 2 lists all the system calls intercepted for propagating taints and establishing an infection graph **in our prototype**.

Category	Function Name
File system access	`NtCreateFile`, `NtOpenFile`, `NtReadFile`, `NtWriteFile`, `NtDeleteFile`, `NtSetInformationFile`
Registry access	`NtCreateKey`, `NtOpenKey`, `NtDeleteKey`, `NtSetValueKey`, `NtDeleteValueKey`, `NtQueryValueKey`
Process, thread and section	`NtCreateProcessEx`, `NtCreateThread`, `NtCreateSection`
Networking	`NtDeviceIoControlFile`

Table 2. System calls intercepted in AGIS kernel monitor.

Using the system calls in Table 2, the kernel monitor propagates taint across different subjects and objects. Specifically, a tainted subject (e.g., a process) taints another subject or object if it creates or modifies the latter; a tainted object taints a subject if the subject reads from it. An exception is that we avoid tainting some critical system service programs such as `LSASS.EXE` which interact with most running processes, as otherwise the whole system would be tainted. This leaves open the possibility to exploit the vulnerabilities of these services to pass infection components. However, since AGIS is designed for honeypots instead of real clients, we always can add extra protections to these services, e.g., anomaly detection in [18].

In order to capture network activities, we parse parameters of the system call `NetDeviceIoControlFile`. Other system calls are used to detect malicious behaviors specified by security policies. For example, the system call `NtUserSetWindowsHookEx` indicates an attempt to hook Windows message-handling mechanism, which has been intensively used by keyloggers [42]. Our prototype took the policies in Table 1 for infection detection. Another application of these calls is to supply the call site information to the static analyzer, which helps anchor the instruction path an executable followed. For this purpose, we need to find out their call sites.

Locations of call sites[6] are found in our prototype through a stackwalk algorithm, which tracks down the stack frames of a system call from the kernel to user thread stack until a return address within the executable's image is identified. A problem is that some executables' routines may not have stack frames at all as a result of compiling optimizations, which stops the stack-walk prematurely. We mitigate this problem using a known technique [37] to guess the address and then verify it by checking the instruction prior to it, which should be a `call`.

3.2 Static Analyzer

The objective of static analysis is to extract the instruction sequences indispensable for the detected malicious calls to happen. To this end, we implemented a static analysis tool to dissemble an executable, build its control flow graph (CFG), identify the execution path with observed calls and chop that path to obtain the instruction sequences.

We implemented a disassembler on the basis of Proview PVDASM [15], an open source disassembling tool. **It builds a CFG for every executable with malicious calls.** From the CFG, the static analyzer identified all execution paths which contain the sites of all the calls we observed, including those responsible for an infection's malicious behaviors recorded in its malicious action set $\mathcal{M}$. These paths are constructed through a backward depth-first search, starting from the site of a malicious call. That call is set to be the last one on the sequence of calls which triggered a security policy, because the execution paths reaching it must also include the ones reaching other malicious calls that happened before it. However, this treatment might not work if the executable is only partially disassembled, which prevents us from finding a complete CFG. When this happens, we simply start the search from every malicious call to find the paths matching as many observed call sites as possible.

Instructions indispensable to malicious calls are discovered through chopping the paths in the set of selected paths (denoted by $\mathcal{P}$) with respect to these calls. The chopping algorithm we implemented focuses more on the dataflow than the control flow, as the latter is easier for the attacker to manipulate using obfuscation techniques. However, it keeps all the structure information such as existence of branches and loops. To find the chop for a call c over a path $p \in \mathcal{P}$, we first identify the `push` instructions related to that call, which can be done using models of API functions. Taking these `push` instructions as chop targets, the static analyzer backtracks p to pick up other instructions on the

[6] The location of a call site is represented by the offset of a call site to the beginning of its residing image and the residing image name.

path carrying elements affecting the return values of c directly or transitively through the `push` instruction. For example, the instruction `push ecx` passes a parameter to c, and its preceding instruction `mov ecx, dword ptr [0x4+2*ebx]`, which sets the value of `ecx`, is also included in the chop.

The call-site information might be too coarse to uniquely identify the path a program executed. This results from the existence of multiple paths between two call sites. When this happens, we end up with multiple chops. Our analyzer first attempts to intersect these chops, which yields a set of instructions guaranteed to be necessary to the chop target. In the worst case, if the resulted intersection is too small, we can keep either all the chops or only those that will not cause a significant false positive, which can be estimated using a set of legitimate executables.

4 Evaluation

In this section, we describe our evaluation of AGIS. Our objectives were to understand its efficacy from three perspectives: (1) effectiveness in detecting new infections, (2) quality of the signatures it generates, and (3) resilience to moderate obfuscations. To this end, we conducted experiments with infections caused by strains of real-world malware and their variants, including MyDoom (D/L/Q/U), NetSky(B/X) [6], Spyware.KidLogger [23], Invisible KeyLogger 97 Shareware version [3], and Home Keylogger v1.60 [2]. **All the malware are collected from Internet, and the experiments were carried out in a VMware installed with Windows XP (service pack 2) on a host with 3.2GHz CPU and 1GB memory.**

4.1 Infection Detection

We ran AGIS against all nine strains of malware inside the VMware and successfully detected all of them. MyDoom (D/L/Q/U) and NetSky (B/X) triggered the mass-mailing rule in Table 1, and all three keyloggers set off the keylogger rule. AGIS automatically generated the infection graphs for all of these infections. Here, we take MyDoom.D and Spyware.KidLogger as two examples to elaborate on our experiments.

MyDoom.D. Mydoom.D is a mass-mailing worm, which is also capable of turning off anti-virus applications, stopping the computer from booting and reducing system security [4]. This worm arrives as an attachment to an email.

Our kernel monitor reported the following behaviors. It first copied itself to `\WINDOWS\SYSTEM32\` as `taskmon.exe` and dropped another executable `shimgapi.dll` to the same directory. Then, it modified many registry keys, including the Run registry key to point to itself. The monitor observed that a thread of the executable invoked a large number of `NtReadFile` calls from the same call site. These calls touched 588 files. This well exceeds the threshold for detecting a search loop, which we set as 100. Another thread of the application made a number of calls to `NtDeviceIoControlFile`, which turned out to be the attempts to invoke `Send`, delivering messages to the SMTP server related to an email address we included in a "goat" html file. So far, the event conditions for mass-mailing worms were unambiguously met and the rule was triggered.

Spyware.KidLogger. Spyware.KidLogger is a spyware program that logs keystrokes. It can also monitor instant messaging, web browsing and the applications activated periodically. Symantec rates its risk impact as high [5].

Within AGIS, `KidLogger` deposited and executed a temporary executable `is-I486L.tmp` which further dropped several files, including executables such as `Hooks.dll`, `_shfoldr.dll`, `MainWnd.exe`, `report.exe`, and `dsk_bmp4.exe`. `is-I486L.tmp` then modified the Run registry key to point to `MainWnd.exe`. After being activated, `MainWnd` had the RunService registry key pointed to itself. Then, it initiated a call to `NtUserSetWindowsHookEx`, the parameters of which indicated that the hook was set for the keyboard, and that the callback function was located inside `Hooks.dll`. That file responded to keystrokes with a number of calls to `NtWriteFile`. Our static analyzer scanned that DLL and found an execution path from the entry of the callback function to the site of the API call which led to `NtWriteFile`. Moreover, the sites of all the calls observed from `hooks.dll` which happened before `NtWriteFile` also appeared on that path. This matched the *ExistPath* condition, and triggered the keylogger rule (in Table 1), which classified `MainWnd`, `hooks.dll` and the KidLogger installer as malicious executables.

Policy False Positives. We ran both security policies on 19 common applications including BiTtorrent, web browsers, Microsoft Office, Google desktop and others. Our prototype did not classify any of them as an infection. Google desktop was found to hook keyboard. However, its hook procedure did not write to files or make network connections. Other applications' behavior did not even come close to the keylogger policy. Some applications such as Outlook were observed to make connections to a mail server. However, they did not read massive files as a mass-mailing worm does. Actually, the legitimate application making the largest number of calls to `NtReadFile` from a unique call site was PowerPoint, which accessed 90 files. In contrast, MyDoom read 588 files in our experiment.

4.2 Signature Generation

AGIS automatically extracted the chops for all the infections we tested. Again, we use MyDoom.D and KidLogger

API Call	Call Site #	Comments
RegSetValueExA	1	Set the Run Registry key to point to Taskmon.exe
ReadFile	1	Scan the file system for email addresses
WS2_32.dll:send	3	Send emails to SMTP servers

Table 3. Malicious Calls in MyDoom.D.

as examples to explain our results.

MyDoom.D. The kernel monitor reported five malicious calls (Table 3) from the main executable of MyDoom, which was renamed as `TaskMon.exe`. With regards to these calls, our static analyzer extracted three chops, one for setting the registry, one for scanning the file system and one for sending emails. Figure 3 illustrates the execution path for scanning, in which the instructions on the chop are highlighted. From that figure we can easily identify the loop for searching directories (on the left) which contains API calls `FindFirstFileA` and `FindNextFileA`, and its embedded loop for reading files (on the right) which uses `CreateFile` to open an existing file, and then continuously reads from that file. Moreover, the chops automatically extracted from other MyDoom worms and NetSky worms have the similar structures.

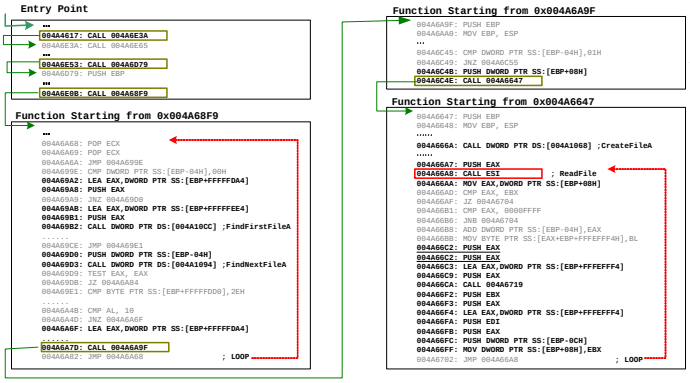

Figure 3. The execution path for scanning email addresses in MyDoom.D. Highlighted instructions are on the chop.

Spyware.KidLogger. We detected five malicious calls from three executables dropped by KidLogger (`KidLogger.exe`, `MainWnd.exe` and `Hooks.dll`). These calls are listed in Table 4. Our static analyzer extracted chops from the recorded calls. Figure 4 demonstrates the execution paths and the chops for `MainWnd.exe` and `Hooks.dll`. `MainWnd.exe` hooked a callback function in `Hooks.dll` to intercept keystrokes. The chop of that DLL unequivocally indicated the malicious mission of the keylogger, which first acquired keystrokes (`GetKeyNameTextA`), and then created or opened a log file (`CreateFileA`) to save them (`SetFilePointer` and `WriteFile`).

Signature False Positives. Two types of signatures were generated from the chops: the regular-expression signature constructed using the approach described in Section 2.3, and the vanilla malware directly built from these chops. To evaluate their false positive rates, we collected `1378` PE

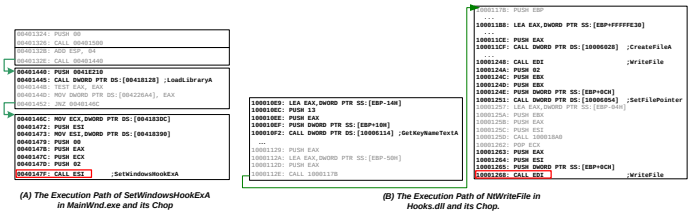

Figure 4. The execution paths and their chops for Spyware.Kidlogger. Highlighted instructions are on the chop.

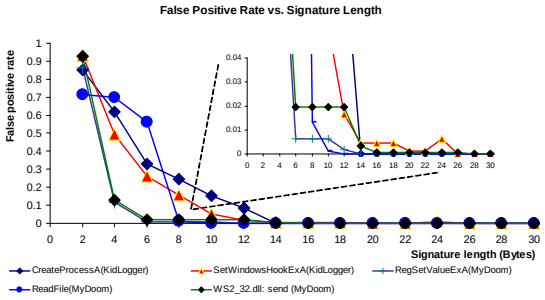

Figure 5. False positive rate vs. signature length.

files from directory '`C:\ProgramFiles`' on Windows XP and used them as a test dataset.

Regular-expression Signatures. A regular-expression signature we used is a conjunction of byte strings which are closest to the site of a malicious call on its chop. Therefore, it is natural to conjecture that the selection of the call affects the quality of the signature. Another important factor related to false positives is signature length. **The longer a signature is, the fewer false positives it will lead to as it is more specific.** The objective of our experiments was to study the relation between these factors and the false positive rate of our signatures. We developed a simple scanner which first took out the executable section of a PE file and then attempted to find the signature from it. Figure 5 describes the experiment results.

In the figure, the signatures constructed from the API functions `RegSetValueExA` and `Send` had the lowest false positive rates. A possible reason is that these functions are less frequently used than the other functions, such as `CreateProcessA`. False positives also decreased with the increase of the signature length. As illustrated by the zoomed subfigure in Figure 5, it was completely eliminated after the length reached `28` bytes.

Vanilla Malware. To evaluate the quality of a vanilla-malware signature, we need to demonstrate that the instruction template (i.e, the chop) we extracted will not appear in a legitimate program. To this end, we developed a static-analysis based scanner which works as follows. It first checks if a program imports all the API functions on the template chop, and then attempts to find an execution path in the program with all these functions on it. If both conditions are satisfied, the scanner further chops that path with regards to an important call, and compares

File Name	API Call	Call Site #	Comments
kidlogger.exe	CreateProcessA	1	Create a process (is-I486L.tmp) to install other code
MainWnd.exe	RegSetValueExA	1	Set the Run Registry key to point to itself
	SetWindowsHookExA	1	Hook the keyboard
Hooks.dll	WriteFile	2	Export keystrokes to a log file

Table 4. Malicious calls of Spyware.KidLogger. Note that there was one temporary executable (is-I486L.tmp) with malicious calls. However, we did not use them because the file was deleted and could not be used for signature generation.

the sequence of the operators of the instructions on the template chop with those on the chop of the normal program. For example, suppose the instructions on the template are `push eax; add eax,ebx; mov ebx,10`; the sequence we attempt to find from the chop in a normal program is `push add mov`.

In our experiment, we scanned all 1378 files, and no false positive was reported by our scanners.

Resilience to Metamorphism. The resilience of AGIS to metamorphic infections was evaluated using a mutation engine based on RPME [17], which can perform three mutation operations: junk code injection, instruction transposition and instruction replacement. To generate metamorphic code, we ran the mutation engine on the execution paths used to extract chops.

RPME performed all three mutation operations on execution paths of MyDoom.D and KidLogger, and then AGIS analyzed them using our static analyzer. As expected, almost all the chops extracted were identical to the original ones except that some adjacent but independent instructions were swapped. This problem does not threaten our approach because **the static analyzer will build a CFG for every instruction sequence, which is used to avoid extracting wrong instructions.** In our experiments, the code size of the execution paths varied from 39 bytes to 467 bytes, while the mutated code kept a constant size of 4K bytes.

Effectiveness against Encode Infections. We also evaluated our prototype using an encoded infection, MyDoom.D which is packed using UPX. In the experiment, our prototype located malicious call sites in the section UPX1 of its executable, and set the attribute of the section to read-only. Rerunning the executable produced an exception which revealed the malicious instruction "`mov [edi],eax`". Our static analyzer chopped the executable using that instruction to generate vanilla malware. Further study shows that the chop extracted actually describes the unpack loop of UPX.

4.3 Performance

We measured the performance of our implementation: infection detection took 73s for MyDoom.D and 66s for KidLogger; signature generation took 60s for MyDoom.D and 6s for KidLogger. As a comparison, Panorama [48] lasts 15 to 25 minutes to detect one malware sample.

5 Discussion

In this section, we will discuss the limitations in the design and implementation of AGIS.

The current design of AGIS could be evaded by the infections in the OS kernel and those capable of countering dynamic analysis. For example, malware can check the SSDT to detect the presence of the kernel monitor and remove its executables. In addition, an infection might deliberately delay running its malicious payload or condition the execution of malicious activities on environmental factors. How to defeat these attacks is part of our future research.

Our current implementation only monitors malware's interaction with operating systems (OS), which are observable from system calls. However, some infections are in the form of add-ons to a legitimate application and their interactions with the application do not go through OS. An example is the spyware based on Brower Helper Object [26]. Our implementation will let these behaviors slip under the radar. This limitation, however, is more to do with the simplicity of the implementation than the weakness of the design. Recent research [26, 29] shows there is no essential technical barrier to wrapping the interactions in AGIS.

Another concern is that these excluded system services in Section 3.1 could be used to conduct malicious activities to evade detection. To this end, we can adopt a technique which traces the subjects and objects indirectly affected by an infection through monitoring the outputs of a service program corresponding to the request from a tainted process.

AGIS generates an infection signature based upon the malicious behaviors observed from suspicious executables. Such an approach, however, may only capture a subset of the infection's activities. We believe the subset of malicious behaviors in many cases is enough to characterize an infection, as what we really care about is not an accurate classification of malware strains but detection of the presence of dangerous code and disruption of its activities.

Detection of metamorphic malware is an undecidable problem in general. That said, theoretically it is possible to develop a metamorphic malware that thoroughly modifies the way it accomplishes its mission for every infection. In fact, many malware authors built their metamorphic or polymorphic malware using the mutation engines developed by third parties [17, 49]. These mutation engines have limited capability to carry out in-depth obfuscations such as injection of junk code related to an important call, which requires understanding of the code being obfuscated. We plan to further investigate this problem to enhance AGIS.

As we discussed before, static analysis has only limited capabilities to handle indirect calls and jumps, which also constrains the effectiveness of AGIS. This problem can be mitigated through dynamic analysis. For example, we can use static analysis to identify branching instructions and then instrument the code before them to help identify their jump targets at runtime. Dynamic slicing techniques can also be applied to extract the chop when obfuscations confound static analysis.

6 Related Work

Techniques for automatic generation of malware signatures have been intensively studied [40, 27, 24, 34, 32, 44, 33, 31, 13, 12, 14, 47, 30]. However, existing research mainly focuses on generation of exploit signatures which reflect the intrusion vectors malware employs to break into a vulnerable system. Such signatures are designed for preventing an exploit, not for detecting an already infected system. Infection signatures are used to detect infections, which serves to complement exploit signatures.

Only limited research has been conducted to automate infection signature generation. The first automatic tool for generating virus signatures was proposed by Kephart and Arnold [22]. Their approach extracts a prevalent byte sequence from infected files which serve as "goats" to attract infection from a virus in a sandboxed environment. This method does not handle metamorphic malware well and heavily relies on the replication property of viruses. By comparison, AGIS can generate signatures for non-replicable infections, and those caused by metamorphic malware. Wang et al. [45] recently proposed NetSpy, a network-based technique for generating spyware signatures. NetSpy intercepts spyware's communication with spyware companies, and extracts prevalent strings from its messages. However, network traffic may not carry sufficient information to characterize an infection. AGIS is a host-based technique, which complements NetSpy with the host information related to an infection's behaviors.

Recently, Kirda et al. proposed a behavior-based spyware detection technique [26, 16] which applies dynamic analysis to detect suspicious communications between an IE browser and its BHO plug-ins, and then analyzes the binaries of suspicious plug-ins to identify the library calls which may lead to leakage of user's inputs. Although this approach shares some similarity with AGIS, it is for detection only, not for signature generation. In addition, it only works on BHO based spyware. In contrast, AGIS is able to work against standalone spyware.

The taint-analysis technique AGIS uses to construct infection graphs resembles those proposed for other purposes such as tracking intrusion steps and recovering a compromised system. BackTracker [25] traces an intrusion back to the point it entered the system. Process Coloring [21] is another system designed for a similar purpose. Back-to-the-Future framework [19] offers a system repair technique to restore an infected system using a log recording infected files and registry entries.

With the objective of malware detection, Panorama [48] tracks how taint information flows among system objects in an instruction level, but it suffers from indirect dependencies, anti-emulation techniques [35], and high performance overhead. In contrast, our approach tracks taint propagation at system-call level. It makes an overestimate in taint propagation and thus be able to largely handle indirect dependencies. Our experimental results shown that such overestimation has not introduced any additional false positives in the detection phase. AGIS alleviates the other two problems in Panorama by running in a real PC environment with reasonable performance overhead, enabling it to run on a production system. Besides detection, our approach also generates infection signatures to detect infected systems.

7 Conclusions

In this paper, we present AGIS, the first host-based technique for automatic generation of infection signatures. AGIS tracks the activities of suspicious code inside a honeypot to detect novel malware, and identify a set of malicious behaviors which uniquely characterizes its infection. Dynamic and static analyses are further used to automatically extract the instruction sequences responsible for these behaviors. A range of infection signatures can be constructed using these sequences, from regular-expression signatures for legacy scanners to vanilla malware for static analyzer [10]. Our empirical study demonstrates the efficacy of the approach.

The current design of AGIS still leaves much to be desired. In the follow-up research, we plan to investigate extension of dynamic analysis to tackle indirect branching, and techniques to counter anti-emulation tricks played by malware executables. In addition, we will develop new scanning techniques which use our signatures to quickly and accurately detect infections.

References

[1] A - z list of all threats and risks. http://www.symantec.com/security_response/threatexplorer/azlisting.jsp?azid=W, as of Feb. 2007.

[2] Home keylogger 1.60. http://www.kmint21.com/keylogger/, as of May 2007.

[3] Invisible keylogger 97 shareware version. http://www.spywareguide.com/product_show.php?id=438/, as of May 2007.

[4] Mydoom.a/d specification. http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99, as of May 2007.

[5] Spyware kidlogger specifications. http://www.symantec.com/security_response/writeup.jsp?docid=2006-020913-4035-99, as of May 2007.

[6] Symantec virus specifications. http://www.symantec.com/security_response, as of May 2007.

[7] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic;, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 154–163, 2006.

[8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[9] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, New York, NY, USA, 2004. ACM Press.

[10] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Usenix Sexurity Symposium*, August 2003.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[12] M. Costa, J. Crowcroft, M. Castro, A. I. T. Rowstron, L. Zhou, L. Zhang, and P. T. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of SOSP*, pages 133–147, 2005.

[13] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of MICRO*, pages 221–232, 2004.

[14] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[15] P. Disassembler. http://pvdasm.reverse-engineering.net/, as of May 2007.

[16] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *To appear in the 2007 USENIX Annual Technical Conference*.

[17] R. P. M. Engine. http://vx.netlux.org/vx.php?id=er10, as of May 2007.

[18] S. Forrest, S. Hofmeyr, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[19] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 257–268, 2006.

[20] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium(WIN-NT-99*, pages 135–144, Berkeley, CA, July 12–15 1999. USENIX Association.

[21] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, and E. H. Spafford. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.

[22] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, 1994.

[23] s. c. KidLogger. Keystroke logger, Web activity monitor. http://www.rohos.com/kid-logger/, as of May 2007.

[24] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, pages 271–286, San Diego, CA, USA, August 2004.

[25] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[26] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of USENIX Security Symposium 2006*, 2006.

[27] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[28] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detection and prevention of stack buffer overflow attacks. *Commun. ACM*, 48(11):50–56, 2005.

[29] Z. Li, X. Wang, and J. Y. Choi. Spyshield: Preserving privacy from spy add-ons. In *RAID*, pages 296–316, 2007.

[30] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, New York, NY, USA, 2005. ACM Press.

[31] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13^{th} *Annual Network and Distributed Systems Security Symposium*, 2006.

[32] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 226– 241, Okaland, CA, USA, May 2005.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, Feburary 2005.

[34] G. Portokalidis and H. Bos. SweetBait: Zero-hour worm detection and containment using honeypots. Technical Report IR-CS-015, Vrije Universiteit Amsterdam, May 2005.

[35] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *ISC*, pages 1–18, 2007.

[36] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, 1995.

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

[38] R. Sekar and P. Uppuluri. Synthesizing fast intrusion detection/prevention systems from highlevel specificationsn. In *Proceedings of USENIX Security Symposium*, page 63C78, 1999.

[39] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[40] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of OSDI*, pages 45–60, 2004.

[41] L. Spitzner. Honeypots: Catching the insider threat. In *Proceedings of ACSAC*, pages 170–181, 2003.

[42] K. Subramanyam, C. E. Frank, and D. H. Galli. Keyloggers: The overlooked threat to computer security. In *1st Midstates Conference for Undergraduate Research in Computer Science and Mathematics*, Oct. 2003.

[43] Symantec. The digital immune system. `http://www.symantec.com/avcenter/reference/dis.tech.brief.pdf`.

[44] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of IEEE INFOCOM*, Miami, Florida, USA, May 2005.

[45] H. Wang, S. Jha, and V. Ganapathy. Netspy: Automatic generation of spyware signatures for nids. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 99–108, 2006.

[46] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring autostart extensibility points (aseps) for spyware management". In *USENIX LISA 2004*, 2004.

[47] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 223–234, New York, NY, USA, 2005. ACM Press.

[48] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[49] M. Z0mbie. `http://vx.netlux.org/vx.php?id=er10`, as of May 2007.