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, a host-based technique that detects infections by 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 an 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., [28, 17, 14, 24, 22, 30, 23, 21, 6, 8, 33, 20, 27]) 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 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 the vulnerable systems. We meet this challenge by exploring the automatic generation of a different type of signature, an *infection signature*, which characterizes malware's behavior when it resides on a system. The main objective of constructing infection signatures is to detect the presence of a malware that has successfully penetrated a system. While an exploit signature can be generated through analyzing the software vulnerability which allows the exploit to happen [33, 2], infection signatures are generally more difficult to get, due to the diversity of malware's behavior in an 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 [13]. A similar approach has been adopted by Symantec in their digital immune system [29]. 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 [3].

There are other malware detectors that identify malware by its MD5 checksum. Generation of a checksum signature can be easily automated. However, it is too specific to accommodate any modification to the code. Wang et al. [31] 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. Consequently, our approach does not begin with detectors for a code-injection at-

1-4244-2398-9/08/$20.00 ©2008 IEEE

tack (e.g., using an input-provided value as a pointer [18]), 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 behaviors 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 [4, 5], 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 behavior-based spyware detection that employs a composite of dynamic and static analyses to detect spyware in the form of browser plug-ins [16, 9]. However, our technique complements that approach in that it works on standalone malware such as keyloggers and mass-mailing worms. Recently, Yin et al. [34] proposed Panorama, a technique that applies instruction-level taint analysis to malware detection and analysis. Panorama is designed for infection detection, whereas AGIS is the first host-based approach for automatic infection signature generation, though it also contains a coarser-grained detection component. Our experimental evaluation shows that AGIS' system-call level taint analysis seems to be sufficient for detecting many existing infections and is much more efficient.

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 (c.f. [16]), our current focus is 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 that 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 and Implementation

In this section, we describe the general design of AGIS and a prototype we implemented under Windows XP. To generate infection signatures, AGIS takes two key steps: *malicious behavior detection* and *infection signature extraction*. We first present the general idea through a simple example, and then elaborate on these individual steps.

2.1 Overview

As an illustrative example, 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. From these calls, AGIS constructs an *infection graph* which records the relations among the Trojan and the files it downloads, as evidenced by, 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 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

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

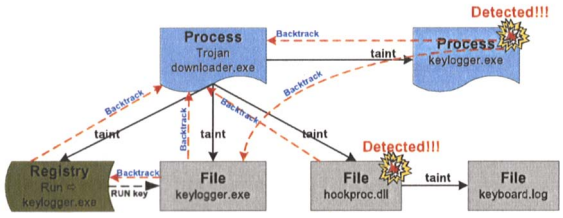

Figure 1. The infection graph of the example. The dotted lines annotated with 'backtrack' describe the backtracking process. The vertices with 'Detected!!!' are detected violating security policies.

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 observed system calls, 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 malware and if so, to identify a set of behaviors which characterize it. AGIS adopts a novel technique which first builds an infection graph to describe the relationships 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. These behaviors are used to generate infection signatures.

Infection Graph. An infection graph is a tuple $\langle V, A \rangle$, where V is a set of vertices and A is a set of arcs. The 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) [32] such as the Run registry key to the executable it points to.

AGIS builds an infection graph using a system call level taint-analysis technique. This was achieved in our implementation through a kernel monitor that hooks Windows system service dispatch table (SSDT) and the shadow table to intercept system calls. The monitor first taints the suspicious code trapped in a honeypot and its process, which forms the first set of vertices on the infection graph we call 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 1 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 detects a sequence of hooking and recording behaviors, and the mass-mailing policy detects reading files and then connecting to SMTP servers. In AGIS, we specify security policies using Behavior Monitoring Specification Language (BMSL) [26]. Table 1 describes two example policies.

A policy can capture many malware instances: e.g., we examined 23 mass-mailing worms reported by Symantec, all of which exhibit the behaviors described above. Our experiments on 19 common applications using the above policies (see Section 3.1) did not report any false positives.

Infection Detection and Behavior Extraction. AGIS detects an infection by matching the behaviors of suspicious code to the event pattern of a security policy. Most such behaviors can be directly observed through system calls, while the rest need 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 for an execution path connecting f to a function exporting a file in the control flow graph (CFG) of a tainted executable.[2] 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 detects an infection and puts the detected processes and their executable files in the infection set $\mathcal{N}$. After that, the *backtrack* function is invoked, which inductively adds to $\mathcal{N}$ all vertices from which $\mathcal{N}$ can be reached in the infection graph. During this process, the file representing a vertex in $\mathcal{N}$, which could also be a vertex, is also included in the infection set. These vertices and their arcs form a subgraph connecting the sources to the behaviors that trigger a security policy. We then remove the vertices which do not have physical representations on the hard disk and their arcs. The remaining subgraph records all the behaviors leading to the ma-

[2] Static analysis can be defeated by anti-disassembling techniques [11], or deep obfuscations of the executable. When this happens, we can use instruction-level dynamic analysis to verify the existence of an execution path.

NAME	SECURITY POLICY	COMMENTS
Keylogger rule	$any()*; hook(keyboard, f) \| f1 = f; any()*; (WriteFile \|\| Sendto) \| ExistPath(f1) \longrightarrow detected(\mathcal{N}) \&\& backtrack(\mathcal{N}) \&\& GenSign(\mathcal{N})$	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	$any()*; (ReadFile() \| ExistSearchLoop; (!Sendto(SMTP))*; Sendto(SMTP) \longrightarrow detected(\mathcal{N}) \&\& backtrack(\mathcal{N}) \&\& GenSign(\mathcal{N})$	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 written in BMSL. 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.

licious activities and retrievable from a compromised system. We call the set of such behaviors the *infection action set*, denoted by $\mathcal{M}$, which is used to generate signatures for executable files in $\mathcal{N}$.

Figure 1 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 by extracting the instruction sequences responsible for an infection's behaviors in its infection action set $\mathcal{M}$. These instruction sequences are extracted through a combination of dynamic and static analysis.

Dynamic Analysis. An executable's behaviors observed by AGIS are in the form of system calls. Our dynamic analyzer implemented in the kernel monitor intercepts these calls and examines their call stacks to find out the return addresses inside 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 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 used to decrypt and run it.

A challenge is that malware can forge a stack frame, including its return address, and enter a library or API call using a jump instruction other than a `call`. If the jump is direct, then this jump to the library or API call can be detected by static analysis and used either in our subsequent static analysis or used within a signature directly (since normal programs do not do this). If the jump is indirect, and if the jump target cannot be computed by static analysis, then dynamic analysis is needed to infer the jump target, either by instrumenting the jump or via instruction-level tracing. A similar approach would be necessary to analyze multi-threaded malware in which one threat manipulates the stack frames of another. AGIS does not implement this analysis at present, however, and refining this approach is a topic of ongoing work.

Static Analysis. After locating all the call sites, our static analyzer, which was implemented based upon Proview PVDASM (`http://pvdasm.reverse-engineering.net/`), applies a chopping technique [25] 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 carrying the influence of one specific instruction (the source criterion) to another (the target criterion) [25]. 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 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 traversed. 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: (1) disassembles the executable's binaries and constructs a control flow graph; (2) finds an executable path p which includes $(c_0, \ldots, c_n)$; (3) for $k = n \ldots 1$, chops the instruc-

tion 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 [4] 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 [4] 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 [4]. The code transposition attack becomes powerless in the presence of the CFG, which restores the original program flow. Register reassignment and instruction substitution are of more concern for infection scanning than signature generation. However, a static-analysis based scanner can convert the output of AGIS to an intermediate form [5] 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.[3] If there is a discrepancy, AGIS reports that the malware 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 that influence it.

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 [1] 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 an execution will produce an exception, which reveals the location of the instruction. We evaluated this approach using a real infection (Section 3.2), and successfully extracted the chop for the decryption loop. However, this approach might identify an incorrect 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.

[3] 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.

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 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 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) (`http://www.symantec.com/security_response`), Spyware.KidLogger (`http://www.rohos.com/kid-logger/`), Invisible KeyLogger 97 Shareware version (`http://www.spywareguide.com/product_show.php?id=438/`), and Home Keylogger v1.60 (`http://www.kmint21.com/keylogger/`). The experiments were carried out in a virtual machine (VMware) installed with Windows XP (service pack 2) on a host with 3.2GHz CPU and 1GB memory.

3.1 Infection Detection

We ran AGIS against the nine strains of malware inside 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 successfully generated 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 (http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99). 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. At this point, 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 (http://www.symantec.com/security_response/writeup.jsp?docid=2006-020913-4035-99).

Within AGIS, KidLogger deposited and executed a temporary executable is-I486L.tmp which further dropped several files, including executables Hooks.dll and MainWnd.exe. is-I486L.tmp then modified the Run registry key to point to MainWnd.exe. After being activated, MainWnd pointed the RunService registry key 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.exe, Hooks.dll and the KidLogger installer as malware.

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 the 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 numerous files as a mass-mailing worm does. 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.

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 2. Malicious Calls in MyDoom.D.

3.2 Signature Generation

AGIS automatically extracted the chops for all the infections we tested. Again, we use MyDoom.D and KidLogger as examples to explain our results.

MyDoom.D. The kernel monitor reported five malicious calls (Table 2) from the main executable of MyDoom, which was renamed as TaskMon.exe. Our static analyzer extracted three chops, one for setting the registry, one for scanning the file system and one for sending emails. Figure 2 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 read from that file. Moreover, the chops automatically extracted from other MyDoom worms and NetSky worms have similar structures.

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 3. Our static analyzer extracted chops from the recorded calls. Figure 3 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 preserved the important instructions 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 files from directory 'C:\ProgramFiles' on Windows XP and used them as a test dataset.

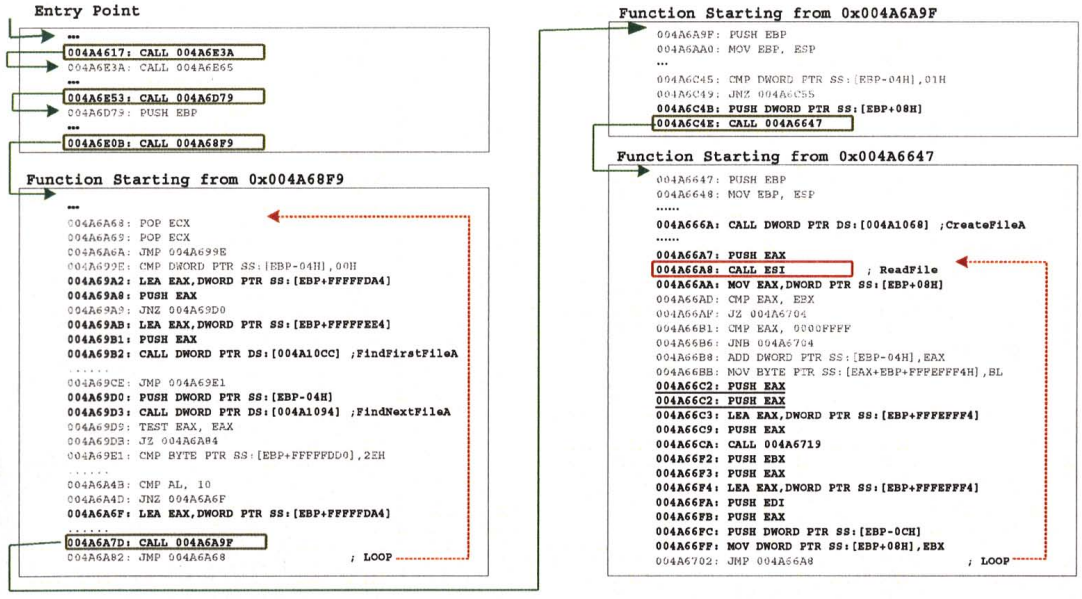

Figure 2. The execution path for scanning email addresses in MyDoom.D. Highlighted instructions are on the chop.

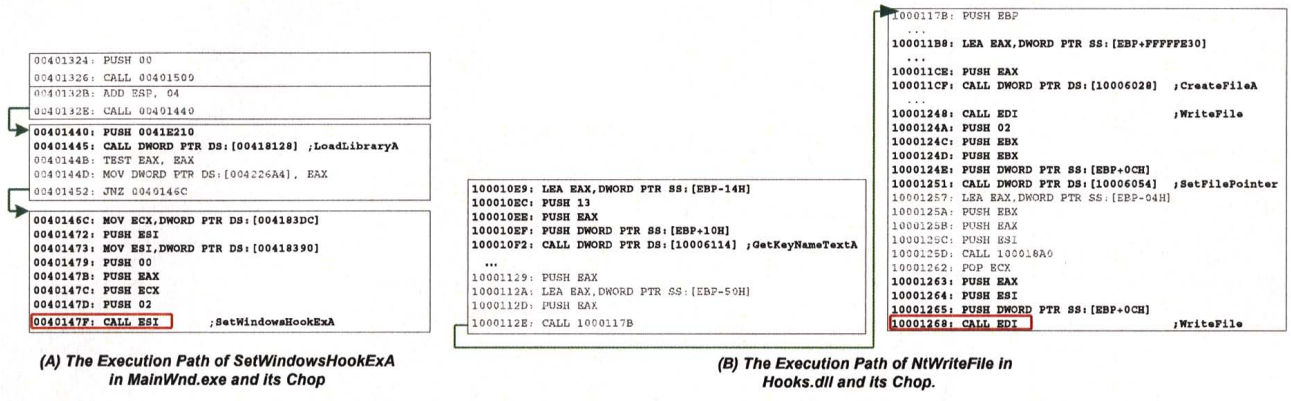

Figure 3. The execution paths and their chops for Spyware.Kidlogger. Highlighted instructions are on the chop.

Regular-expression Signatures. The 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 more specific it becomes and therefore the fewer false positives it will introduce. 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 4 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

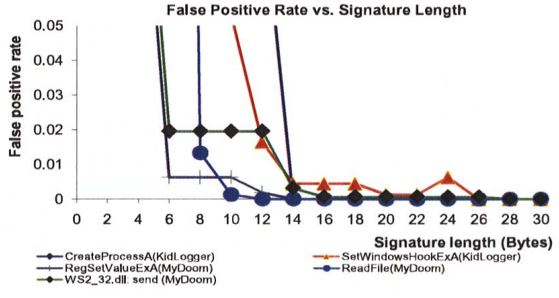

Figure 4. False positive rate vs. signature length.

with the increase of the signature length. As illustrated in Figure 4, false positives were 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

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 3. 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.

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 regard to the last call within the path, and compares 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 ability of AGIS to withstand metamorphic malware was evaluated using a mutation engine based on RPME (`http://vx.netlux.org/vx.php?id=er10`), which can perform three mutations: 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 the execution paths of MyDoom.D and KidLogger, which were subsequently analyzed using our static analyzer. As expected, all the chops extracted were identical to the original ones except that some adjacent but independent instructions were swapped. We believe that this problem is minor because it will not cheat our analyzer into extracting the instructions unrelated to malicious behaviors. Moreover, chopping also reveals the dependency relations among individual instructions, which can be used to identify the instruction sequences differing only in the positions of independent 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.

3.3 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 [34] takes 15 to 25 minutes to detect one malware sample.

4 Discussion and Limitations

The current design of AGIS could be evaded by malware that penetrates the operating system (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.

Our current implementation only monitors malware's interaction with the OS, which are observable from system calls. However, some infections are in the form of add-ons to a legitimate application and so their interactions with the application does not go through the OS (e.g., spyware based on Brower Helper Objects [16]). Our implementation will let these behaviors slip under the radar. While our implementation will not detect such malware, recent research [16, 19] suggests there is no essential technical barrier to wrapping the interactions in a technology like AGIS.

Dealing with metamorphic malware is a challenge for AGIS that we are continuing to explore. Theoretically it is possible to develop a metamorphic malware that thoroughly modifies the way it accomplishes its mission for every infection. In practice, however, many malware authors build their metamorphic or polymorphic malware using the mutation engines developed by third parties. As discussed in Section 2.3, AGIS is tolerant of several obfuscations common to such tools.

As discussed in Section 2.3, the AGIS implementation is limited in its ability to identify the locations of API calls by malware that forges return addresses in its stack frames and performs these calls using indirect jumps. This problem can be mitigated through dynamic analysis. For example, we can use static analysis to identify indirect jumps 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.

5 Related Work

Techniques for automatic generation of malware signatures have been intensively studied [28, 17, 14, 24, 22, 30, 23, 21, 7, 6, 8, 33, 20]. 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 [13]. 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-replicating infections, and is tolerant to some forms of metamorphic malware. Wang et al. [31] 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. In contrast, 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 [16, 9] which applies dynamic analysis to detect suspicious communications between an IE browser and its Browser Helper Object 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, its focus is BHO-based spyware, versus the standalone spyware that AGIS targets.

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 [15] traces an intrusion back to the point it entered the system. Process Coloring [12] is another system designed for a similar purpose. Back-to-the-Future [10] 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 [34] tracks how taint information flows among system objects at an instruction level. In contrast, our approach tracks taint propagation at a coarser granularity (system calls) and so potentially overestimates taint propagation. Our experimental results shown that such overestimation has not introduced any additional false positives in the detection phase. Moreover, our coarser approach enables AGIS to run with less performance overhead. Besides detection, our approach also generates infection signatures. MetaAware [35] describes an approach to identify metamorphic malware by extracting and matching code patterns that are used to execute system calls. Proposed independently, the signature generation step of AGIS is similar to the code pattern extraction step of MetaAware. Compared to MetaAware, our approach can detect unknown malware, while our approach doesn't focus on an algorithm to match signatures. The two approaches can complement each other.

6 Conclusions

In this paper, we presented AGIS, a host-based technique for automatic generation of infection signatures. AGIS tracks the activities of suspicious code inside a honeypot to detect malware, and identifies a set of malicious behaviors that characterizes the infection. Dynamic and static analyses are 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 a static analyzer [4]. Our empirical study demonstrates the efficacy of the approach.

Acknowledgements

This work was supported in part by the National Science Foundation Cyber Trust program under Grant No. CNS-0716292.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] R. Sekar and P. Uppuluri. Synthesizing fast intrusion detection/prevention systems from highlevel specificationsn. In *Proceedings of USENIX Security Symposium*, pages 63–78, 1999.

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

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *ACSAC*, pages 411–420, 2007.