Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis

Heng Yin[*†] Dawn Song[‡*] Manuel Egele, Christopher Kruegel, Engin Kirda[§]

hyin@ece.cmu.edu *dawnsong@cs.berkeley.edu* {*pizzaman,chris,ek*}*@seclab.tuiwen.ac.at*

[*] Carnegie Mellon University, Pittsburgh, PA, USA
[†]College of William and Mary, Williamsburg, VA, USA
[‡] UC Berkeley, Berkeley, CA, USA
[§] Technical University Vienna, Austria

ABSTRACT

Malicious programs spy on users' behavior and compromise their privacy. Even software from reputable vendors, such as Google Desktop and Sony DRM media player, may perform undesirable actions. Unfortunately, existing techniques for detecting malware and analyzing unknown code samples are insufficient and have significant shortcomings. We observe that malicious information access and processing behavior is the fundamental trait of numerous malware categories breaching users' privacy (including keyloggers, password thieves, network sniffers, stealth backdoors, spyware and rootkits), which separates these malicious applications from benign software. We propose a system, Panorama, to detect and analyze malware by capturing this fundamental trait. In our extensive experiments, Panorama successfully detected all the malware samples and had very few false positives. Furthermore, by using Google Desktop as a case study, we show that our system can accurately capture its information access and processing behavior, and we can confirm that it does send back sensitive information to remote servers in certain settings. We believe that a system such as Panorama will offer indispensable assistance to code analysts and malware researchers by enabling them to quickly comprehend the behavior and innerworkings of an unknown sample.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Malware Detection, Malware Analysis, Dynamic Taint Analysis, Spyware

1. INTRODUCTION

Malicious software (i.e., Malware) creeps into users' computers, collecting users' private information, wrecking havoc on the Internet and causing millions of dollars in damage. Surprisingly, even software provided by reputable vendors may contain code that performs undesirable actions which may violate users' privacy. For example, Google Desktop, a popular local file system search tool, actually sends sensitive user information such as the local search index files back to Google's servers in certain configuration settings [18]. In another widely publicized example, Sony Media Player installs a rootkit without the user's knowledge in order to enforce copyright restrictions and sends back users' music listening habits [34].

Malware detection and analysis is a challenging task, and current malware analysis and detection techniques often fall short and fail to detect many new, unknown malware samples. Current malware detection methods in general fall into two categories: signature-based detection and heuristics-based detection. The former cannot detect new malware or new variants. The latter are often based on some heuristics such as the monitoring of modifications to the registry and the insertion of hooks into certain library or system interfaces. Since these heuristics are not based on the fundamental characteristics of malware, they can incur high false positive and false negative rates. For example, many benign software access and modify registry entries. Hence, just because an application creates hooks in the registry does not mean that it is malicious (i.e., the application could be a useful system utility). Furthermore, to evade detection, malware may attempt to hook library or system call interfaces that the detector does not monitor. Even worse, since many rootkits hide in the kernel, most such heuristics-based detectors cannot detect them as they do not necessarily modify any visible registry entries or library or system call interfaces.

In this paper, we propose a novel approach for the detection and analysis of privacy-breaching malware. We observe that numerous malware categories, including spyware, keyloggers, network sniffers, stealth backdoors, and rootkits, share similar fundamental characteristics, which lies in their malicious or suspicious information access and processing behavior. That is, they access, tamper, and (in some cases) leak sensitive information that was not intended for their consumption. For example, when a user inputs some text into an editor, benign software (except the editor) will not access this text, whereas a keylogger will obtain the text, and then send it to the attacker. This behavior is typically exhibited without the user's knowledge or consent and it is this fundamental trait that separates such malicious applications from benign software.

Thus, based on this observation, we have designed and developed an end-to-end approach to automatically identify this fundamental trait of malicious/suspicious information access and processing behavior of a given program. At a high level, our approach is a three-step process: test, monitor, and analyze. When examining a malware sample, we first load it into our analysis environment and run a series of automated tests on it. Each test generates events that introduce sensitive information into the system in a way that is not destined for the sample under analysis. For example, the introduced information may be keystrokes that are intended for the Windows login process, or user input that is entered into web forms. We then monitor the behavior of the sample during the tests and record its information access and processing behavior. Finally, we automatically analyze the recorded information access and processing behavior of the sample to detect malicious/suspicious behavior and use the behavioral information we extract from the sample for detailed analysis.

To monitor and record the information access and processing behavior of the sample in the test cases, we propose to use *whole-system, fine-grained taint tracking*. The approach works by marking the sensitive information introduced in the tests as tainted, and monitoring taint propagation over the whole system (including the propagation through the kernel and all applications). We monitor the taint propagation at the hardware level. To perform meaningful analysis, we also need a mechanism to extract operating-system level information. For example, we need to know which processes and which program modules operate on tainted data, or which files the tainted data is written to. We call this concept *operating-system-aware* taint analysis.

By combining the taint propagation information at the hardware level with operating-system-level knowledge, we then generate *taint graphs*. A taint graph is a representation of information flow that shows the processes that access tainted data, how the data propagates through the system, and finally, to which file or network connection this data is written to. Based on taint graphs, we can define various policies that specify the characteristic behavior of different types of malware. By checking the policies against the taint graph of an unknown sample, we can then enable automatic detection and analysis of malicious code from numerous categories.

To explore the feasibility of our approach, we have designed and developed an end-to-end prototype called Panorama. Our experiments demonstrate that Panorama is successful in detecting all malicious code samples in our test set, generating only a small number of false positives. During the tests, we also observed that fine-grained taint analysis suffers from a significant performance degradation (a slowdown by a factor of 20). However, since Panorama is targeted to support off-line detection and analysis of malware, and since optimization is not our main focus while building the prototype, we believe that although significant, this overhead is not a severe limitation for our purposes. We also believe that the approach we propose can be used in combination with existing malware crawlers (e.g., such as [25]) to search the web for unknown malware.

In summary, this paper makes the following contributions:

- We observe that a fundamental trait of privacy-breaching malware lies in their information access and processing behavior to sensitive information, and propose an end-to-end automatic approach to classify and detect malware using their information access and processing behaviors. Our approach does not rely on signatures and thus, it can detect novel instances of malicious code. And since it captures the fundamental trait of malware, it provides a unified approach to detect and analyze a wide spectrum of different malware.

- We have designed and developed Panorama, an end-to-end system that can automatically analyze samples for malicious information access and processing behavior. As a critical component of Panorama, we have designed and developed a whole-system, fine-grained, operating-system-aware, dynamic taint tracking system to enable us to monitor and investigate the unknown sample's information access and processing behavior to sensitive information.

- In our extensive experiments, our system detected all the malware samples and had very few false positives. The malware samples include a wide range of different classes of malware, such as keyloggers, password sniffers, packet sniffers, stealth backdoors, rootkits and spyware. Using the Google Desktop as a case study, we demonstrate that our system accurately captures its information access and processing behavior, and that we can confirm by automated analysis that it does leak sensitive information to remote servers.

The paper is structured as follows: The next section gives an overview of our approach. Section 3 presents details on the design and implementation of Panorama. Section 4 discusses our taint graph-based malware analysis and detection. Section 5 presents the experimental results. Section 6 discusses the potential evasions and our countermeasures. Section 7 surveys related work and Section 8 concludes the paper.

2. OVERVIEW OF APPROACH

Given an unknown program to analyze, we wish to automatically determine whether it exhibits malicious information access and processing behavior. At a higher level, our approach to automatically detect whether an unknown sample exhibits malicious behavior is a three-step process: test, monitor, and analyze. In this work, we focus on the analysis of Windows-based malware. Hence, we use an out-of-the-box installation of Microsoft Windows as the analysis environment. We regard all code that comes with this installation as being *trusted* (in contrast to the unknown sample about which we have no information). We load the sample to be analyzed into this environment and mark which files belong to the loaded sample. We then run the entire environment including Microsoft Windows and the loaded

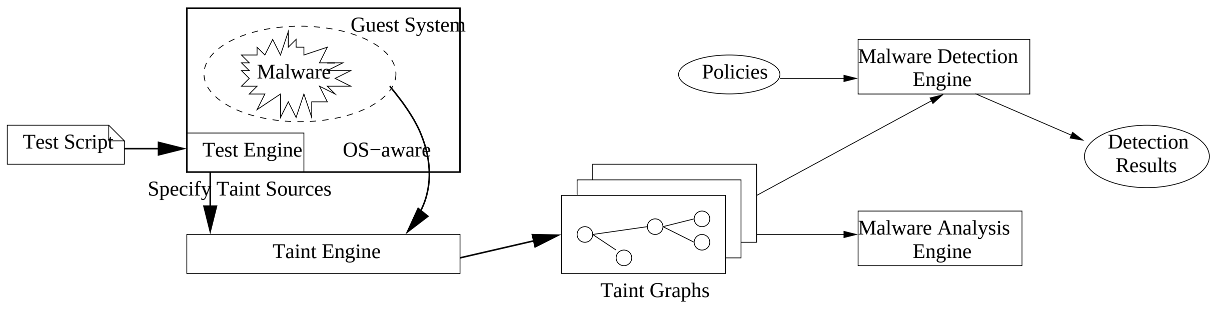

Figure 1: System Overview

sample in our system Panorama. Figure 1 depicts the architecture of Panorama. The system consists of the taint engine, the test engine, the malware detection engine, and the malware analysis engine.

To perform our automatic malware detection and analysis, we run a series of automated tests, which is performed by the *test engine*. For each test, we generate events that introduce sensitive information into the guest system. This sensitive data is sent to some trusted application, and is not destined for the sample that is under analysis. We then monitor the behavior of the sample during the tests and record its information access and processing behavior with respect to the sensitive information introduced in the tests. To this end, we have designed the *taint engine*, which performs whole-system, fine-grained information flow tracking. It monitors how the sensitive information propagates within the whole guest system (including the propagation through the kernel and all applications). In particular, we need to investigate whether the information has propagated into the sample (i.e., whether it has been accessed by the sample) and what the sample has done with the information (e.g., sending it to an external server via the network).

Note that even though dynamic taint analysis has been proposed before, our approach is the first generic framework that applies dynamic taint analysis to the problem domain of detecting and analyzing privacy-breaching malware. Furthermore, our system offers several new capabilities that are necessary in our problem setting: (1) Our system is *OS-aware*—in addition to hardware-level taint tracking, we need to understand the high-level representations of hardware states for the analysis; (2) We also need to identify what actions are performed by or on behalf of the sample under analysis, even if the sample performs code unpacking and dynamic code generation, and executes actions through libraries, etc.; (3) Our monitoring needs to be whole-system and fine-grained, in order to precisely detect all actions of the sample.

The system-wide information behavior is captured by a graph representation, which we call *taint graph*. Taint graphs capture the taint propagation from the initial taint source (i.e., the sensitive information introduced in the tests) throughout the system. Using taint graphs, we can determine whether the unknown sample has performed malicious actions. In general, the decision whether an information access and processing behavior is considered malicious or benign is made with the help of policies. One characteristic property of many types of malicious code (such as keyloggers, spyware, stealth backdoors, and rootkits) is that they steal, leak or tamper with sensitive user information. Consider the following examples: (1) The user is typing input into an application such as a Microsoft Notepad, or is entering his user name and password into a web login form through a browser, while an unknown sample also accesses these keystrokes; (2) The user is visiting some websites, while an unknown sample accesses the webpages or URLs and sends them to a remote host; (3) The user is browsing a directory or searching a file, while an unknown sample intercepts the access to the directory entries and tampers with one or more entries. We devise a set of policies, which are used by the *malware detection engine* to detect malware from unknown samples. Finally, since taint graphs present invaluable insights about the samples' information access and processing behaviors, analysts can use the *malware analysis engine* to examine the taint graphs, for detailed analysis information. More information on taint-graph-based analysis and detection is provided in Section 4.

3. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of Panorama. First, we describe the hardware-level taint tracking in Section 3.1. Then we discuss the mechanisms that can map hardware-level operations (such as instructions executed on the processor) to the corresponding operating-system objects (such as processes), in Section 3.2. Finally, we describe our approach to performing automated testing and generating taint graphs in Section 3.3.

3.1 Hardware-level Dynamic Taint Tracking

To perform whole-system, fine-grained taint tracking, we need to monitor how tainted data propagates throughout the whole system including the OS and the applications. Since the source code for commodity software such as the Windows operating system and applications are usually not available, we choose the approach of dynamic instrumentation—i.e., we monitor the whole system execution in a processor emulator and dynamically instrument code to keep track of how tainted data propagates during program execution. We choose to implement Panorama on QEMU [29, 3], a generic and open source processor emulator, because of its efficiency (achieved through dynamic translation and caching) when compared to previous processor emulators such as Bochs [5].

Our hardware-level taint tracking is similar in spirit to a number of previous systems [10, 26, 13, 35, 12]. However, since our goal is to enable whole-system fine-grained taint analysis, our design and implementation is the most complete. For example, previous approaches either operate on a single process only [12, 26, 35], or they cannot deal with memory swapping and disks [10, 13].

Shadow Memory.

We use a shadow memory to store the taint status of each byte of the physical memory, CPU's general-purpose registers[1], the hard disk and the network interface buffer. Each tainted byte is associated with a small data structure storing the original source of the taint and some other book keeping information (which is necessary for generating taint graphs). The shadow memory is organized in a page-table-like structure to ensure efficient memory usage. With the shadow memory for the hard disks, the system can continue to track the tainted data that has been swapped out. Obviously, this also enables the tracking of the tainted data that has been saved to a file and is then read in.

Taint Sources.

All sensitive information that is introduced into the system in the automated tests is marked as a taint source. Panorama supports taint input from hardware, such as the keyboard, network interface, and hard disk. Tainting a high-level abstract data object (e.g. the output of a function call, or a data structure in a specific application or the OS kernel) would also be appropriate. Note that taint sources have to be specified as close to the hardware (i.e., low-level) as possible. For example, tainting the input typed at the keyboard level is better than tainting the input in a browser form. Otherwise, malware may try to evade detection by creating hooks that are invoked before the input arrives at the browser.

Taint Propagation.

After a data source is tainted, we need to monitor each CPU instruction and DMA operation that manipulates this data in order to determine how the taint propagates. For data movement instructions and DMA operations, the destination will be tainted if and only if the source is tainted. For arithmetic instructions, the result will be tainted if and only if any byte of the operands is tainted. We also handle the following special situations.

Constant function: some instructions or instruction sequences always produce the same results, independent of the values of their operands. A good example is the instruction "`xor eax, eax`" that commonly appears in IA-32 programs as a compiler idiom. After executing this instruction, the value of `eax` is always zero, regardless of its original value. We recognize a number of such special cases and untaint the result.

Table lookup: a tainted input may be used as an index to access an entry of a table. The taint propagation policy above will not propagate taint to the destination, because the value that is actually read is untainted. Unfortunately, such table lookup operations appear frequently, such as for Unicode/ASCII conversion in Windows. Thus, we augmented our propagation policy with the following rule: if any byte used to calculate the address of a memory locations is tainted, then, the result of a memory read using this address is tainted as well.

Control flow evasion: the taint information may also propagate through control flow. The following example illustrates this situation.

```
switch(x) {
  case 'a': y='a'; break;  case 'b': y='b'; break; ...
}
```

Note that the above code fragment copies the value of variable `x` to `y`, without propagating the taint status. That is, `y` will always be untainted, even when `x` is tainted.

The situation outlined above occurs rarely in regular code. However, it does appear in the keystroke handling routines in Windows 2000 and later versions. In our experiments with Windows XP, we observed that the Unicode characters derived from keystrokes were not tainted as expected. After reviewing the raw taint propagation events and examining the Windows kernel code using IDA Pro [22], we determined that taint tracking stops at a keystroke Unicode conversion routine called `_xxxInternalToUnicode` (which is in part of the `win32k.sys` system file). Interestingly, Chow et al. faced the same problem in their TaintBochs [10]. Unfortunately, they did not have a solution. The translation of scancode into corresponding unicode characters involves a loop that contains a switch statement such as the example discussed previously. We solved the problem by specially instrumenting an instruction within the function `_xxxInternalToUnicode`. This instrumentation checks the taint status of the input parameter of the function, and appropriately propagates the taint status to its output parameter.

Being aware of this property, malicious code may exploit control flow evasion in the future to cut off the taint flow in order to thwart detection. The current implementation of Panorama does not handle this situation. This does not cause problems for now, because to the best of our knowledge no existing malware has used this technique. Furthermore, we will incorporate the static analysis approach proposed in [14] into the future implementation of Panorama to prevent this potential evasion.

3.2 OS-Aware Taint Tracking

Resolving process and module information.

When an instruction is operating on tainted data, we need to know which process and module this instruction comes from. In some rare situations, instructions may also be dynamically generated and executed on the heap.

Maintaining a mapping between addresses in memory and modules requires information from the guest operating system. To obtain this information, we developed a kernel module called *module notifier*. We load this module into the guest operating system to collect the updated memory map information. The module notifier registers two callback routines. The first callback routine is invoked whenever a process is created or deleted. The second callback routine is called whenever a new module is loaded and gathers the address range in the virtual memory that the new module occupies. In addition, the module notifier obtains the value of the CR3 register for each process. As the CR3 register contains the physical address of the page table of the current process, it is different (and unique) for each process. All the information described above is passed on to Panorama through a predefined I/O port.

[1]For the sake of simplicity, in the current implementation, flags, debug registers, control registers and SIMD (e.g. MMX and SSE) registers are not considered. However, adding the necessary tracking for these registers would be straightforward.

Since our module notifier component resides in the guest operating system, malicious code may attempt to tamper with it. For example, malware could attempt to send incorrect information to the predefined I/O port or tamper with the code image of the module. To ensure the authenticity of the messages that Panorama receives from the module notifier, we check the program counter of the instruction that is responsible for sending this message. Of course, only instructions that belong to the module notifier are permitted to send messages. We also protect the integrity of the code of the module notifier by marking the corresponding memory region read-only. As a result, any attempts to tamper with the code of the module notifier can be detected and prevented. Note that a more secure approach to resolving process and module information is to directly examine the process and module objects from the outside. The disadvantage of this approach is less of portability. That is, different versions of Windows, and even different service packages, need be handled differently. Thus, we decided to use the first, more portable approach in our proof-of-concept prototype implementation.

Resolving filesystem and network information.

In addition to mapping instructions executed on the processor to operating-system processes, we are also interested in obtaining more information when data is exchanged between the memory and hardware devices. In particular, we are interested in more details about when tainted data is written to the hard disk or sent over the network. More precisely, when tainted data is written to the hard disk, we wish to identify which file it is written to. Analogously, when tainted data is transmitted over the network, we would like to know which TCP (or UDP virtual) connection it is sent over or received from.

We integrated a disk forensic tool called "The Sleuth Kit" (TSK) [36] into Panorama for gathering filesystem information. Specifically, when tainted data is written to a block on the hard disk, TSK can determine which file this block belongs to. In addition, when a file on disk is selected as a taint source, TSK will identify all data blocks that belong to this file (so that all blocks can be appropriately tainted). The toolkit achieves these goals by scanning and parsing the on-disk meta-data structures.

Resolving network information is straightforward. When tainted data is sent out, we simply check the packet header to find out which connection it belongs to.[2] Similarly, when selectively tainting the incoming traffic of a specific connection, we check its packet header and taint the packet accordingly. Tainting incoming network packets from the network card is performed at the granularity of (virtual) connections.

Identifying the code under analysis and its actions.

An important task of our system is to identify the actions of the code under analysis. In particular, we are interested in observing cases in which the potential malware sample accesses tainted data. It is clear that the code under analysis operates on tainted data if an instruction in it accesses the taint directly. This can be checked in a straightforward fashion by consulting the mapping between instruction addresses and modules. However, there are two important cases in which it is not the malicious sample itself that accesses tainted data, but code that operates on its behalf.

The first case occurs when the sample under analysis dynamically generates new code (either by decrypting data regions, or by generating code on the fly). In this case, the derived code belongs to the sample under analysis, but the origin of the code is not reflected in our module mapping. To handle this situation, we taint the complete code segment of the sample under analysis, using a special label. Whenever an instruction is executed that is marked with the special label, the output of this instruction receives the special label as well. This strategy helps identify all code regions derived from the original sample, such as uncompressed and decrypted instructions from packed executables, or those dynamically generated.

The second case occurs when the given code calls a piece of trusted code in order to perform tainted operations on its behalf. In this case, the program counter would point to the trusted code, and we would miss the potential malicious behavior of the given sample, if we only look at the program counter. We use the following observation to identify taint propagation that is performed by trusted system modules on behalf of the malware: Whenever the malicious code calls a trusted function to propagate tainted data, the value of the stack pointer at the time of the function call must be greater than the value of the stack pointer at the time when the tainted data is actually propagated. This is because one or more stack frames have to be pushed onto the stack when making function calls, and the stack grows toward smaller addresses on the x86 architecture.

Based on our observation, we use the following approach to identify the case when trusted functions propagate tainted values on behalf of the code under analysis: Whenever the execution jumps into the code under analysis (or code derived from it), we record the current value of the stack pointer, together with the current thread identifier. When executing jumps out of this code, we check whether there is a recorded stack pointer for the current thread identifier, and if so, whether this value is smaller than the current stack pointer. If this is the case, we remove the record as the code is not on the stack anymore. Whenever a trusted module propagates tainted data, we check whether there is a recorded stack pointer under the current thread identifier. If so, we consider this tainted data being propagated by the code under analysis. Note that the current thread identifier is mapped into a well-known virtual address in Windows. Hence, obtaining its value is straightforward.

Note that the strategy described above will detect all taint-related action on behalf of the malicious code, given that they are performed in the same thread context. While this is true most of the time, there are cases in which the actual taint propagation occurs in an asynchronous fashion. For example, when the code calls an API function asynchronously to save the tainted data to a file, the API function immediately returns to the caller. The actual action that is requested is performed later. We have identified several kernel API calls (dealing with filesystem and network access) that may be used asynchronously. When such a function is invoked, we analyze the stack pointers to determine whether both the code under analysis is calling this function and the input buffer is tainted. If this is the case, we

[2]We may not be able to obtain transport-layer information directly from IP fragments. In the current prototype implementation, we do not solve this infrequent case. However, re-assembling the fragments and extracting this information is quite straightforward if desired.

treat this tainted buffer as being propagated by the analyzed sample.

3.3 Automated Testing and Taint Graph Generation

3.3.1 *Automated Testing*

The test engine in Panorama allows us to perform the analysis of samples and the detection of malicious code without human intervention. It executes a number of test cases that mimic common tasks that a user might perform, such as editing text in an editor, visiting several websites, and so on. The specific test cases used in our experiments will be discussed in Section 4.1. To automatically run tests, our test engine is equipped with scripts that execute all steps necessary for each test case. For our current implementation, these scripts are based on the open source program AutoHotkey [1]. Scripts can be either manually written or automatically generated by recording user actions while a task is performed.

Whenever the test engine executes a certain test case, it introduces input (such as keystrokes or network packets) into the system. To determine which part of this input should be tainted (and with which taint label), the test engine cooperates with the taint engine. Currently, our system defines the following nine different types of taint sources: *text*, *password*, *HTTP*, *HTTPS*, *ICMP*, *FTP*, *document*, and *directory*, which will be discussed in Section 4.1. For example, when editing a document in an editor, the test engine asks the taint engine to send keystrokes to the editor, and label them as *text*; when entering password in a secure web form, the test engine asks the taint engine to send keystrokes and label them as *password*. When considering these cases, it becomes evident that the taint engine requires support from the test engine to properly label input. In both cases, the keystroke information enters the system. However, in the former case, the keystroke is considered *text* as it is sent to the one of the text editors. In the latter, the recipient of the input is a password field and the keystroke information is marked as *password*. Clearly, this information is test-specific and not available at the hardware level. The data received as a response to the web requests are tainted as *HTTP*. The packets received in response to ping requests are labeled *ICMP*. The information sent by the FTP server are marked *FTP*. Finally, when listing a directory, all accessed disk blocks that hold file directory information are tainted as *directory*. The communication between the test engine and the taint engine is via an intercepted registry writing API: the test engine writes information into a predetermined registry entry, and taint engine intercepts this API call and then obtains the information.

3.3.2 *Taint Graph Generation*

The system-wide propagation of tainted input introduced by the test engine forms a graph over the processes/program modules and OS resources. For example, assume that a keystroke is tainted as *text* because it is part of the input sent to a text editor. When a user process A reads the character that corresponds to the keystroke, this fact is recorded by linking the *text* taint source to process A. When this process later writes the character into a file F, from where it is then read by process B, we can establish a link from process A to the file, and subsequently from file F to process B. For clarity, we generate one graph for each taint source with a different label (that is, one graph that shows the flow of data labeled as *text*, one for *password*, . . .). For each taint source, the taint propagation originating from this source forms a directed graph. We call this graph a *taint graph*.

More formally, a taint graph can be represented as $g = (V, E)$, where V is a set of vertices and E is a set of directed edges connecting the vertices, and we use $g.root$ to represent the root node of graph g (i.e., the taint source). A vertex can either represent an operating system object (such as a process or module), an OS resource (such as a file), or a taint source (such as keyboard or network input with the appropriate labels). An edge between two vertices v_1 and v_2 is introduced when tainted data is propagated from the entity that corresponds to v_1 to the entity that corresponds to v_2.

When generating the taint graphs, the taint engine maps the hardware-level taint propagation information to operating-system level. For example, the taint engine determines which process and which module (such as which `dll`) has performed a certain operation, and it also keeps track of whether this operation is performed on behalf of the sample under analysis. Also, writes to disk blocks are attributed to file objects and network operations to specific network connections. To further simplify the taint graphs, we apply the following optimizations, without losing the dependencies between the sample under analysis and other objects: (1) we make the vertices for system kernel modules transparent; (2) for user-level instructions, if they are not derived from the sample under analysis (i.e., they are trusted), they are attributed to the processes they are running in, instead of the modules they are from.[3]

In a taint graph, each vertex is labeled with a (type, value) pair, where value is the unique name that identifies the vertex. For the root node, the type is one of the nine different input taint labels introduced previously. For any non-root node, the type represents the category of the node as a OS object, including process, module, keyboard, network, and file. Formally, the type of a vertex can be defined in a hierarchical form, as follows:

```
type ::= taint_source | os_object
taint_source ::= text | password | HTTP | HTTPS| FTP
     | ICMP | document | directory
os_object ::= process | module | network | file
```

Figure 2 shows an example of a taint graph. We use ellipses to represent process nodes and use shaded ellipses to represent the module node. We use an octagon to represent the taint source (here, a password typed on the keyboard), and a rectangle to represent the other nodes. We will give more description of this graph in Section 4.2.

4. TAINT-GRAPH-BASED MALWARE DETECTION AND ANALYSIS

In this section, we describe how taint graphs can be used to detect malware, and how they help to understand the actions of malicious code,

4.1 Taint-Graph-Based Malware Detection

[3]In other words, the presence of a module node in a taint graph indicates at least one instruction of this module stems from the sample.

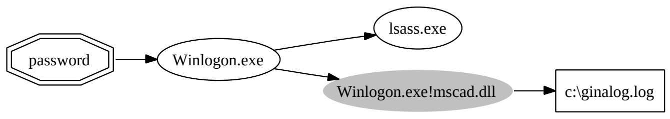

Figure 2: An example of taint graph. This graph reflects the procedure for Windows user authentication. While a password thief is running in the background, it catches the password and saves them to its log file "c:\ginalog.log".

Test case description	Introduced inputs
1. Edit a text file and save it	text, document
2. Enter password in a GUI program	password
3. Log in a secure website	URL, password, HTTPS
4. Visit several websites	URL, HTTP
5. Log into an FTP server	text, password, FTP
6. Recursively list a directory	directory
7. Send UDP packets into the system	UDP
8. Ping a remote host	ICMP

Table 1: The test cases and introduced inputs.

.

Our essential observation is that numerous types of malicious code, including keyloggers, password thieves, network sniffers, stealth backdoors, spyware/adware, and rootkits, exhibit anomalous information access and processing behavior. Currently, we categorize three kinds of anomalous behavior: *anomalous information access*, *anomalous information leakage*, and *excessive information access.*

Anomalous information access behavior.

For some information sources, a simple access performed by the samples under analysis is already suspicious. We refer to this behavior as *anomalous information access behavior.*

Considering the keyboard inputs, such information sources may include the text input sent to the text editor, the command sent to the command console, and the passwords sent to the Windows Logon dialog and secure web pages. Benign samples do not access these inputs, whereas *keyloggers* and *password thieves* will access these inputs. Keyloggers refer to the malicious programs that capture keystrokes destined for the other applications, and thus will access all these inputs. Password thieves, by definition, steal the password information, and therefore will access the password inputs. Note that password thieves can be a subset of keyloggers, because keyloggers may also record passwords.

Similarly, some network inputs are not supposed to be accessed by unknown samples. For example, ICMP is designed for network testing and diagnosis purpose, and hence only operating system and trusted utilities (e.g. `ping.exe`) use it. For many TCP and UDP applications, the incoming TCP and UDP traffic can only be accessed by their own and the operating system. Benign samples do not interfere with the process of these inputs. However, *network sniffers* and *stealth backdoors* access these inputs for different purposes. Network sniffers eavesdrop on the network traffic to obtain valuable information. Even though a network sniffer may not be directly interested in these inputs, it usually has to access them to check if they are valuable. Stealth backdoors refer to a class of malicious programs that contact with remote attackers without explicitly opening a port. To achieve stealthiness, the stealth backdoors either use an uncommon protocol such as ICMP, create a raw socket, or intercept the network stack, in order to communicate with remote adversaries. The ICMP-based stealth backdoors will access ICMP traffic. The raw-socket-based stealth backdoor will access all the packets with the same protocol number. For example, a TCP raw socket will receive all TCP packets. The stealth backdoors intercepting the network stack will behave like a network sniffer.

Anomalous information leakage behavior.

For some other information sources, it is acceptable for the samples to access them locally, but unacceptable to leak the information to third parties. For example, *spyware/adware* programs record users' surfing habits and send this private information to third parties. In contrast, benign BHOs (i.e., Browser Helper Objects) may access this information but will not send it out. We consider the following as information leakage: the sample under analysis accesses the information and then saves it to disk or sends it over the network. Note that saving the information to disk covers three situations: saving it to files, the registry, and even individual disk blocks. We consider information sources like HTTP, HTTPS, documents, and URLs fall into this category.

Excessive information access behavior.

For some information sources, benign samples may access some of them occasionally, while malicious samples will access them excessively to achieve their malicious intent. We refer to it as anomalous information excessive access behavior.

The directory information is such a case. *Rootkits* exhibit excessive access behavior to the directory information, because they attempt to conceal their presence in the filesystem by intercepting the accesses to directory information and removing the entries that point to their files. Thus, when recursively listing directories, we will see the rootkit samples accessing many disk blocks that contain directory information. A benign program may access some directory entries, or even scan directories occasionally. However, it is very unlikely that it accesses the same directories at the same time while we list directories.

Test cases and policies.

According to the above discussion, we compile the following test cases and introduce the inputs with corresponding labels, as shown in Table 1. Specifically, we introduce *text*, *password*, *URL* inputs from the keyboard, *HTTP*, *HTTPS*, *FTP*, *ICMP*, and *UDP* inputs from the network, and *document* and *directory* input from the disk. Note that in the test case 6, to eliminate the possibility that a benign program scans the same directory at a different time, we clean the taint labels of the visited directory entries after finishing with listing the directory. After finishing all the test cases, the test engine waits for a while (a configurable parameter) and then shuts down the guest machine.

From the above discussion, we specify the following policies: (1) *text*, *password*, *FTP*, *UDP* and *ICMP* inputs cannot be accessed by the samples; (2) *URL*, *HTTP*, *HTTPS* and *document* inputs cannot be leaked by the samples; (3) *directory* inputs cannot be accessed excessively by the samples. More formally, we show how these policies are enforced on the taint graphs:

$$\forall g \in G, (\exists v \in g.V, v.type = \texttt{module}) \wedge$$
$$g.root.type \in \{\texttt{text}, \texttt{password}, \texttt{FTP}, \texttt{UDP}, \texttt{ICMP}\}$$
$$\rightarrow Violate(v, \text{“}\texttt{No Access}\text{''}) \qquad (1)$$

$$\exists g \in G, (\exists v \in g.V, v.type = \texttt{module}) \wedge$$
$$(g.root.type \in \{\texttt{URL}, \texttt{HTTP}, \texttt{HTTPS}, \texttt{document}\}) \wedge$$
$$(\exists u \in descendants(v), u.type \in \{\texttt{file}, \texttt{network}\})$$
$$\rightarrow Violate(v, \text{“}\texttt{No Leakage!}\text{''}); \qquad (2)$$

$$(\forall g \in G,\ g.root.type = \texttt{directory} \rightarrow$$
$$\exists v \in g.V, v.type = \texttt{module})$$
$$\rightarrow Violate(v, \text{“}\texttt{No Excessive Access}\text{''}) \qquad (3)$$

In addition to manually specifying the policies, it is possible to automatically generate policies by using machine learning techniques. First, we can gather a representative collection of malware and benign samples as our training set. Using this training set, Panorama will extract the corresponding taint graphs. Then, we need to develop a mechanism to transform a taint graph into a feature vector. Based on the feature vectors for the benign and malicious samples, standard classification algorithms can be applied to determine a model. Using this model, novel samples can then be classified. We will further explore this approach in our future work.

4.2 Taint-Graph-Based Malware Analysis

Given a taint graph, the first step is to check this graph for the presence of a node that corresponds to the sample under analysis. If such a node is present, we obtain the information that the sample has accessed certain tainted input data. This is already suspicious, because the test cases are designed such that input data is sent to trusted applications, but never to the sample under analysis. Once we determine that a sample has accessed certain input, the sample's successor nodes in the graph can be examined. This indicates what has been done with the data that was captured. Such insights can be instrumental for system administrators and analysts to understand the behavior and actions of malware.

As an example, recall the taint graph previously shown in Figure 2. This taint graph has been produced by automatically analyzing the behavior of the password thief program GINA spy [16]. Note that the entered password is received by the Windows Logon process (`Winlogon.exe`). This process passes the password on to `lsass.exe` for subsequent authentication. Interestingly, the password data is also accessed by the sample under analysis (`mscad.dll`), which is loaded by `Winlogon.exe`. This code module reads the password and saves it to a file called `c:\ginalog.log`. The graph correctly reflects how the user password is processed by Windows, and how the password thief intercepts it. In Section 5.2, we discuss a more complex real-world example that we investigated during our experiments.

5. EVALUATION

In this section, we present details on the experimental evaluation of our Panorama system. Our evaluation consisted of three parts. First, we investigated the effectiveness of our taint-graph-based malware detection approach using a large body of real-world malware and benign samples. Then, by using Google Desktop as a case study (i.e., a sample from a vendor whose privacy policy we believed we could trust), we explored the amount of detailed information that we could extract from the taint graph of an unknown sample. Third, we performed tests to evaluate the performance overhead of our prototype. In all our experiments, we ran Panorama on a Linux machine with a dual-core 3.2 GHz Pentium 4 CPU and 2GB RAM. On top of Panorama, we installed Windows XP Professional with 512M of allocated RAM.

Category	Total	FNs	FPs
Keyloggers	5	0	-
Password thieves	2	0	-
Network sniffers	2	0	-
Stealth backdoors	3	0	-
Spyware/adware	22	0	-
Rootkits	8	0	-
Browser plugins	16	-	1
Multi-media	9	-	0
Security	10	-	2
System utilities	9	-	0
Office productivity	4	-	0
Games	4	-	0
Others	4	-	0
Sum	98	0	3

Table 2: Summary of detection results against malware and benign samples.

5.1 Malware Detection

Our malware collection consisted of 42 real-world malware samples, including 5 keyloggers, 2 password thieves, 2 network sniffer, 3 stealth backdoors, and 22 spyware BHOs, and 8 rootkits. Some of these samples were publicly available on the Internet (e.g., from web sites such as *www.rootkit.com*), while others were collected from academic researchers and an Austrian anti-virus company. Furthermore, we downloaded 56 benign, freely-available samples from a reputable and trustworthy web site (*www.download.com*). These benign samples were freeware programs from a wide range of different application domains (such as browser plug-ins, system utilities, and office productivity applications), with the size up to 3MB.

To further facilitate the experiments, we developed a tool using Python to run the samples and automatically perform the installation procedure (if required) using several heuristics. The tool can handle 70% of the samples in our test set. For the remaining samples, some required manual configuration (they were all malware samples), and the others were not properly handled by the heuristics. We then manually installed the remaining samples. We installed up to 3 samples each time. After that, we ran the test cases. We set the test engine to wait for 5 minutes before shutting down the guest machine. Depending on the installation delay, the whole procedure lasts 15 to 25 minutes.

Table 2 summarizes the results of this experiment. We can see that Panorama was able to correctly identify all malware samples, but falsely declared three benign samples to be malicious.

Two of these false positives were personal firewall programs. The third false positive was a browser accelerator.

By checking the taint graphs related to these three samples, we observed that the information access and processing behaviors of these benign samples closely resemble that of malware. In fact, the two personal firewalls install packet filters and monitor all network traffic. Hence, their behavior resembles that of a malicious network sniffer. In the case of the browser accelerator, we observed that the application prefetches web pages on behalf of the browser and stores them into its own cache files. This behavior resembles that of spyware that monitors the web pages that a user is surfing. The reason for our false positives is that our taint-graph-based detection approach can only identify the information access and processing behavior of a given sample, but not its intent. In real-life, the taint graphs are invaluable for human analysts, as they help them to quickly determine and understand whether an unknown sample is indeed malicious, or whether it is benign software that is exhibiting malware-like behavior.

5.2 Malware Analysis

In order to determine how well we are able to perform detailed analysis on an unknown sample, we chose Google Desktop for a case study. This application claims in its privacy policy [19] that it will index and store data files, mail, chat logs, and the web history of a user while the user is working on her system. Furthermore, if the special configuration setting "Search Across Computers" is enabled, Google Desktop will securely transmit copies of the user's index files to Google servers. Hence, Google Desktop, in fact, exhibits some malware-like behavior, as the index files may contain sensitive information about a user (e.g., a list of web sites that the user has visited), and these files are sent to an external server.

First, we downloaded the installation file (*GoogleDesktopSetup.exe*). Before installing the tool, we marked the installation file such that we could track which components would be installed into the system. After the installation was complete, we observed that 18 executables and shared libraries, as well as a dozen data files were installed.

Second, we ran the test cases, using the default settings of Google Desktop (in which "Search Across Computers" is disabled). After performing the test cases, we observed that some components extracted from the installation file accessed the tainted inputs, including HTTPS, HTTP and document. All of this information was later saved into the index files in the local installation directory. To determine if the information is sent out to remote hosts, we kept the system alive for 12 hours. However, we did not observe this behavior.

Third, we changed the settings of Google Desktop and enabled the feature "Search Across Computers". Then, we ran the test cases again and kept the system alive for another 30 minutes. It was evident from the generated taint graphs that, in this mode, Google Desktop did leak the collected information via HTTPS connections to Google servers. We picked a representative taint graph, which clearly illustrates how the components of Google Desktop process the incoming traffic of an HTTP connection from the QEMU web site we visited, (see Figure 3).

By examining this taint graph, we can draw several conclusions: (1) the incoming web page was first received and processed by the Internet Explorer (`IEXPLORE.EXE`), which later saved the content into a cache file (`qemu[1].htm`) under the temporary Internet file folder; (2) a component from Google Desktop (`GoogleDesktopAPI2.dll`) was loaded into the `IEXPLORE.EXE`, obtained the web page, and passed it over to a stand-alone program also from Google Desktop (`GoogleDesktopIndex.exe`); (3) `GoogleDesktopIndex.exe` further processed this information and saved it into two data files (`rpm1m.cf1` and `fiih.ht1`) in its local installation directory; and (4) it sent some information derived from the web page to a remote Google server (72.14.219.147) through an HTTPS connection.

With the capability provided by Panorama, we could confirm that Google Desktop really sends some sensitive information if a special feature is activated (as it also claims in its privacy policy).

5.3 Performance Overhead

We measured Panorama's performance overhead using several utilities in Cygwin, such as *curl*, *scp*, *gzip*, and *bzip2*. When running these tools, we tainted file and network inputs accordingly. We found that the current unoptimized implementation of Panorama suffers a slowdown of 20 times on average. Since Panorama aims to support off-line malware detection and analysis, we believe that this overhead is not a severe limitation for our intended application scenarios. When one considers that unknown malware samples are currently mostly analyzed manually, it is clear that an automated system such as Panorama significantly simplifies and speeds up this task. Also, note that some research has been done to explore more efficient means for dynamic taint analysis. Ho et. al. proposed *Demand Emulation*, in which a running system dynamically switches between virtualized and emulated execution, and emulation is only used when tainted data is being processed by the CPU [20]. Exploring finer-grained hardware protection provided by ECC may further improve the performance significantly [30]. Recently, Qin et. al. explored several optimizations on dynamic binary instrumentation to minimize the run-time overhead [31].

6. DISCUSSION

In this section, we discuss several potential evasion techniques that malware writers may attempt to use in order to thwart the current implementation of Panorama. Furthermore, we discuss the countermeasures that we can employ.

Breaking the propagation of taint information.

As mentioned in Section 3.1, a malware author can attempt to design his code such that the taint engine fails to properly keep track of tainted information. For example, by exploiting indirect dependencies (dependencies encoded using control flow decisions), a malicious program could conceal the fact that sensitive information is leaked. This is a limitation of our current implementation. We will enhance the implementation to keep track of taint propagation via control flow in the future, as in our earlier implementation [14]. Moreover, it is important to note that the current system observes all instances in which the sample under analysis *accesses* tainted data. That is, a malware sample can only hide the fact that it leaks information (as well as the operating system resources that this information is written to). Fortunately, the mere fact that sensitive data is accessed without authorization is often enough to classify a sample as malware.

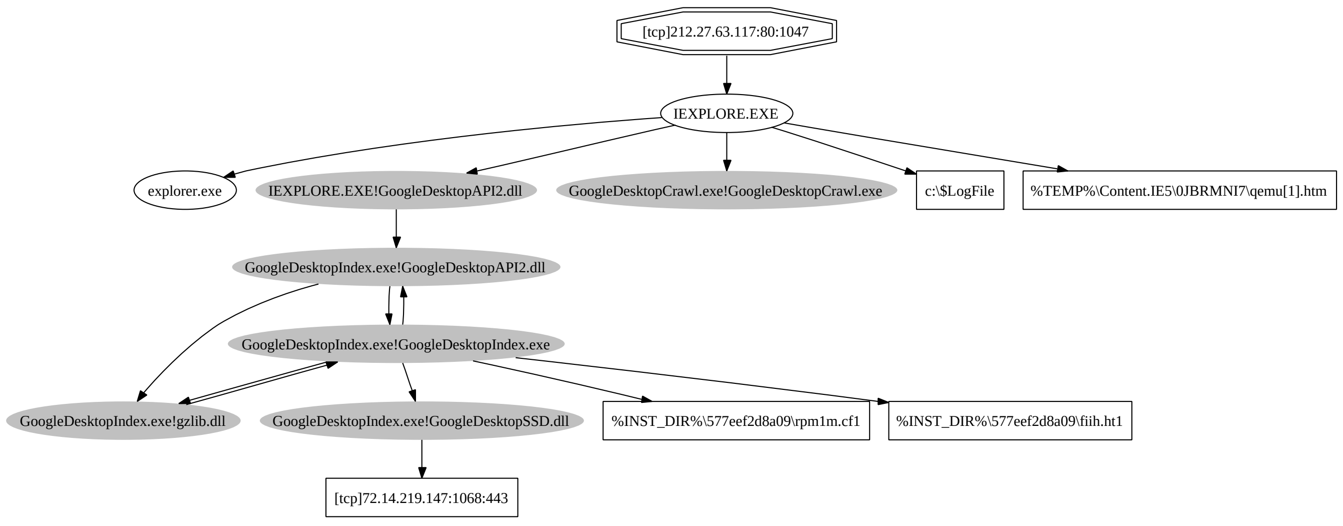
[tcp]212.27.63.117:80:1047
IEXPLORE.EXE
explorer.exe
IEXPLORE.EXE!GoogleDesktopAPI2.dll
GoogleDesktopCrawl.exe!GoogleDesktopCrawl.exe
c:\$LogFile
%TEMP%\Content.IE5\0JBRMNI7\qemu[1].htm
GoogleDesktopIndex.exe!GoogleDesktopAPI2.dll
GoogleDesktopIndex.exe!GoogleDesktopIndex.exe
GoogleDesktopIndex.exe!gzlib.dll
GoogleDesktopIndex.exe!GoogleDesktopSSD.dll
%INST_DIR%\577eef2d8a09\rpm1m.cf1
%INST_DIR%\577eef2d8a09\fiih.ht1
[tcp]72.14.219.147:1068:443

Figure 3: A taint graph that presents how Google Desktop handles incoming web pages. Here, %INST_DIR% represents "c:\Program Files\Google\Google Desktop Search", and %TEMP% is "c:\Documents and Settings\user\Local Settings\Temporary Internet Files".

Not behaving maliciously when tested.

Malware may evade detection by simply not performing malicious behavior while the test cases are conducted. It may stay inactive until certain conditions are satisfied. For example, time bombs activate themselves only on specific dates, and some keyloggers only record keystrokes for certain applications or windows. Malware may also detect if it is running in the QEMU environment and remains dormant if indeed. Ferrie discussed the technique to detect the virtual machines and emulators including QEMU [15]. Our current prototype will not detect this kind of malware. However, some complementary work has been done to address this problem. Vasudevan et al. proposed several stealthy techniques, such that the analysis environment cannot be easily detected [37]. Moser et al. [24] and Brumley et al. [7, 6] also used QEMU to built malware analysis systems, which are able to uncover hidden behavior of malware by exploring multiple execution paths. Incorporating these techniques into our system will be our future work.

Subverting Panorama.

As an emulated environment, Panorama provides strong isolation such that it is unlikely for the malware running inside to interfere with Panorama and the host system. Although it is usually true, some study shows the possibility of subverting the entire emulated environment by exploiting buffer overflows and integer bugs [27]. This problem can be solved by fixing these bugs.

7. RELATED WORK

Malware detection approaches.

Signature based malware detection has been in use for years to scan files on disk and even memory for known signatures. Although semantic-aware signature checking [11] improves its resilience to polymorphic and metamorphic variants, the inherent limitation of the signature based approach is its incapability of detecting previously unseen malware instances. Its usefulness is also limited by the rootkits that hide files on disk and, as demonstrated in Shadow Walker [9], may even hide malware footprints in memory.

Behavior based malware detection identifies malicious programs by observing their behaviors and system states (i.e., detection points). By recognizing deviations from "normal" system states and behaviors, behavior based detection may identify entire classes of malware, including previously unseen instances. There are a variety of detections that examine different detection points. Strider GateKeeper [39] checks auto-start extensibility points in the registry to determine surreptitious restart-surviving behaviors. VICE [8] and System Virginity Verifier [33] search for various hooks that are usually used by rootkits and the other malware. Behavior based detection can be defeated, either by exploring stealthier methods to evade the known detection points, or by providing misleading information to cheat detection tools. In addition, current detection tools usually reside together with malicious programs, and therefore expose to complete subversion. In contrast, our system overcomes these three weaknesses. First, it captures the characteristic information access and processing behavior of malware, and thus cannot be easily evaded. Second, it detects malware based on the hardware-level knowledge and makes very few assumption at software level, and hence cannot be easily cheated. Third, it is implemented completely outside of the victim system, and so strongly protected from being subverted.

The cross-view based rootkit detection technique (e.g. Blacklight [4], Rootkit Revealer [32], and Strider Ghostbuster [2]) identifies hidden files, processes, registry entries by comparing two views of the system: the upper-level view is derived from calling common APIs, while the low-level view is obtained from system states in the kernel or from hardware if applicable. In comparison, our approach for rootkit detection has two advantages: (1) the cross-view based technique requires enumerating all files and registry entries, etc. to find hidden entries, which often takes several hours, whereas our approach only takes a few minutes; (2) the result given by the cross-view based technique can only identify a list of hidden entries, while our approach recognizes the rootkit directly.

Dynamic Taint Analysis.

Dynamic taint analysis has been applied to solve and analyze other security related problems. Many systems [26, 13, 28, 12, 35] detect exploits by tracking the data from untrusted sources such as the network being misused to alter the control flow. Chow et al. made use of whole-system dynamic taint analysis to analyze how sensitive data are handled in operating systems and large programs [10]. The major analysis was conducted in Linux, with source code support of the kernel and the applications. Egele et al. also utilized whole-system dynamic taint analysis to examine BHO-based spyware behavior [14]. Vogt et al. extended the JaveScript engine with dynamic taint analysis to prevent cross-site scripting attacks [38]. Our system is independently developed with OS-aware analysis for closed-source operating systems, and devises a unified machinery for detecting malware from several different categories.

Information flow analysis.

Our system works by analyzing taint graphs to identify suspicious information access and processing behavior of foreign code. This is related to previous work that performs forensic analysis based on information flows. For example, some systems track the flow of information between operating system processes to perform intrusion analysis [23], intrusion recovery [17], and malware removal [21]. However, these systems typically monitor the system call interface and thus, are not as comprehensive and do not provide the same level of precision as our technique. Another limitation of previous systems is that it is often not possible to precisely track data while it is processed by a program. This can introduce incorrect connections between data objects or lead to missed information flows. Also, previous systems do not apply to kernel-mode attacks. Thus, we believe that by performing whole-system, fine grained taint tracking, our method provides higher accuracy than previous work, and we can also handle kernel attacks.

8. CONCLUSION

Malware has brought along serious security and privacy threats. However, existing techniques for malware detection and analysis are ineffective. In this paper, we have proposed *whole-system fine-grained taint analysis* to discern fine-grained information access and processing behavior of a piece of unknown code. This behavior captures the intrinsic characteristics of a wide-spectrum of malware, including keyloggers, password sniffers, packet sniffers, stealth backdoors, BHO-based spyware, and rootkits. Thus, the detection and analysis relying on it cannot be easily evaded. To evaluate the effectiveness of this approach, we have designed and developed a system, called Panorama. In the experiments, we have evaluated 42 malware samples and 56 benign samples. Panorama yields zero false negative and very few false positives. Then we use Google Desktop as a case study. We have demonstrated that Panorama can accurately capture its information access and processing behavior, and we confirm that it does send back sensitive information to remote servers. We believe that a system such as Panorama will offer indispensable assistance to malware analysts and enable them to quickly comprehend the behavior and inner-workings of malware.

9. ACKNOWLEDGEMENT

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422. Partial support is also provided by the International Technology Alliance, and by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and under grant DAAD19-02-1-0389 through CyLab at Carnegie Mellon. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, or the U.S. Goverment or any of its agencies. This work is also supported in part by the Korean Ministry of Information and Communication (MIC) and the Korean Institute for Information Technology Advancement (IITA) under program [2005-S-606-02, Next Generation Prediction and Response Technology for Computer and Network Security Incidents]. Moreover, this work was supported by the Austrian Science Foundation (FWF) under grant P18157, the FIT-IT project Pathfinder, and the Secure Business Austria competence center.

10. REFERENCES

[1] AutoHotkey. `http://www.autohotkey.com/`.

[2] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 368–377, June 2005.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[4] Blacklight. `http://www.europe.f-secure.com/exclude/blacklight/`.

[5] Bochs: The open source IA-32 emulation project. `http://bochs.sourceforge.net/`.

[6] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical Report CMU-CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.

[7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. *Botnet Analysis*, chapter Automatically Identifying Trigger-based Behavior in Malware. 2007.

[8] J. Butler and G. Hoglund. VICE–catch the hookers! In *Black Hat USA*, July 2004. `http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf`.

[9] J. Butler and S. Sparks. Shadow walker: Raising the bar for windows rootkit detection. In *Phrack 63*, July 2005.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security'03)*, August 2004.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In

Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland'05), May 2005.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, October 2005.

[13] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, December 2004.

[14] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)*, June 2007.

[15] P. Ferrie. Attacks on virtual machine emulators. Symantec Security Response, December 2006.

[16] GINA spy. `http://www.codeproject.com/useritems/GINA_SPY.Asp`.

[17] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles(SOSP'05)*, October 2005.

[18] Google's desktop search red flag. `http://www.internetnews.com/xSP/article.php/3584131`.

[19] Google Desktop - Privacy Policy. `http://desktop.google.com/en/privacypolicy.html`.

[20] A. Ho, M. Fetterman, C. Clark, A. Watfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys 2006*, April 2006.

[21] 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 *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[22] The IDA Pro Disassembler and Debugger. `http://www.datarescue.com/idabase/`.

[23] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 223–236, October 2003.

[24] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland'07)*, May 2007.

[25] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proceeding of the 13th Network and Distributed System Security (NDSS'06)*, February 2006.

[26] 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 (NDSS'05)*, February 2005.

[27] T. Ormandy. An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments. `http://taviso.decsystem.org/virtsec.pdf`.

[28] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006*, April 2006.

[29] Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[30] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, February 2005.

[31] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, December 2006.

[32] Rootkit revealer. `http://www.sysinternals.com/Files/RootkitRevealer.zip`.

[33] J. Rutkowska. System virginity verifier: Defining the roadmap for malware detection on windows systems. In *Hack In The Box Security Conference*, September 2005. `http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt`.

[34] Sony's DRM Rootkit: The Real Story. `http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html`.

[35] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

[36] The Sleuth Kit (TSK). `http://www.sleuthkit.org/sleuthkit/`.

[37] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *Proceedings of 2006 IEEE Symposium on Security and Privacy (Oakland'06))*, may 2006.

[38] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, February 2007.

[39] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for spyware management. In *Proceedings of the Large Installation System Administration Conference (LISA'04)*, November 2004.