Whole-system Fine-grained Taint Analysis for Automatic Malware Detection and Analysis

Heng Yin
hyin@cs.wm.edu
College of William and Mary

Dawn Song
dawnsong@cmu.edu
Carnegie Mellon University

Abstract

As malware is becoming increasingly sophisticated and stealthy, effective techniques for malware detection and analysis are imperative. Previous detection mechanisms are insufficient. Signature-based detection cannot detect new malware, and watch-point based behavioral detection can be evaded by stealthier design. Most previous analysis mechanisms are too coarse-grained to capture malware behavior and fail to address kernel-level attacks. We propose *whole-system fine-grained taint analysis* for automatic malware detection and analysis, and build a prototype called TaintQemu. By tainting data from hardware inputs and monitoring its propagation, TaintQemu generate taint graphs. The taint graph represents how information propagates during the system execution. We demonstrate that such whole-system fine-grained taint analysis can capture the intrinsic properties of many different classes of malware and thus offer effective methods for automatic malware detection and analysis. Our evaluation using a wide spectrum of real-world malware demonstrates that our system is effective in detecting many different classes of malware including keyloggers, backdoor, etc., and offer indispensable assistance to system administrators and analyzers for better understanding of the behavior and consequences of malware. also used to automatically detect a wide spectrum of malware, by checking the violations from the normal patterns in taint graph.

1 Introduction

Malicious software (i.e., Malware) is becoming increasingly prevalent and sophisticated. They creep into users' computer systems in ever more creative ways: They could be mistakenly installed when a user clicks on an attachment containing a virus, or visits a malicious website which installs malicious software in the background, or unknowingly installed when the user installs a free-ware or bundled-ware containing spyware or adware. More surprisingly, even software provided by reputable vendors could contain code that performs undesired actions such as leaking users' private data. For example, Google Desktop, a popular local file system search tool, has been reported to send users' private information back to Google's servers [15]. In another example, SONY Media Player has been reported to send users' listening behavior such as which songs the user has listened to back to SONY [35]. Thus, as users and computers cannot live in isolation from the rest of the Internet and foreign code gets downloaded and installed unknowingly or knowingly to the local system all the time, the users and computers are completely oblivious to what code is actually installed on the local system and whether they will have malicious or undesired behavior or consequences. When a piece of code is installed and executed, how can we detect whether it will have certain malicious or undesired behavior such as violating user's privacy? This is an important open research question.

Traditional methods are insufficient in addressing this problem. First, previous approaches on sandboxing or access control do not apply well in this scenario. For example, system utility programs such as Google Desktop need to access user's files and may need to communicate back to Google servers for updates, etc. Users' privacy is only violated when users' private information is sent back to Google. Thus, the traditional sandboxing or access control model is too rigid for this scenario to be able to detect the undesired behavior and at the same time allow Google Desktop to perform its alleged functions.

Second, previous malware detection mechanisms are insufficient. Previous malware detection mechanisms mostly fall into two categories: signature-based detection and watch-point based behavioral detection. The first category, signature-based detection suffers from the drawback that it cannot detect new malware since it needs to know the signature first, and it can be defeated by polymorphic and metamorphic malware variants.

The second category, watch-point based behavioral detection monitors specific points in the computer system and detects malicious code based on certain heuristics. An example is Microsoft's Strider Gatekeeper [38], which monitors auto-start extensibility points to determine surreptitious restart-surviving behavior. A main drawback for this approach, however, is that it needs to know what detection points should be watched and what behavior at that detection point should be monitored. Malware authors keep exploring innovative techniques to achieve their malicious intent and also conceal their presence without going through the previously identified detection points, and thus render the existing detection points useless. For example, early malware detection monitors userland APIs and system calls to detect malware behavior that go through these APIs and system calls. To evade detection, more sophisticated malware moves into the operating system kernel such as kernel rootkit. When detection also moves to the kernel to identify malicious activities, such as detecting kernel hooks [6] and kernel data object manipulation [31], much stealthier malware design has recently been invented to circumvent the detection mechanisms [33]. Its demonstration, a stealthy backdoor called *deepdoor* achieves the backdoor functionality by only modifying a few DWORDs in NDIS data block, (which is supposed to be modified), and eventually evades all the existing detection tools. Another recent study shows the viability of building malware completely out of the victim operating system, leveraging the technique of virtual machines [22]. Thus, the history of the arms race between the malware writers' innovations and the watch-point based behavioral detection mechanisms demonstrate that we need a holistic approach to prevent the malware to evade monitoring and once-for-all put such an arms race to an end.

To address the above issues, we propose a new approach for automatic malware detection and analysis: whole-system fine-grained taint tracking. By monitoring the whole system (including the operating system), malware cannot evade our detection by avoiding previously identified detection points as in the watch-point based behavioral detection methods. Moreover, by analyzing the common traits of malware, we identify a unified approach that enables the detection and analysis of a wide spectrum of malware: fine-grained taint tracking. One fundamental trait of malware lies in its data access. While benign software usually accesses the data of its own interest, malware is inclined to monitor, intercept, and modify the data belonging to other programs and even the operating system. Such malware may exhibit two types of anomalies in its data access and information propagation pattern: (1) Access the information not supposed to be accessed. For instance, keystroke loggers capture keystrokes belonging to other processes, and packet sniffers monitor or intercept all incoming and outgoing packets; (2) Access the information in an eccentric way, to circumvent security mechanisms enforced on the system. For example, to circumvent the firewall, a backdoor may access incoming packets at a lower layer of the network stack. Thus, by monitoring taint tracking in a fine-grained manner, we enable automatic detection and analysis of a wide-spectrum of malware based on these insights.

In particular, we monitor system execution and enable taint tracking at the instruction level. The result from taint tracking forms a *taint graph*. The taint graph represents how information gets propagated. We then show that the taint graph can be used in three ways for automatic malware detection and analysis. First, we can build a policy engine to enforce invariants/properties on the taint graph—taint flow violates these invariants/properties would indicate an attack. . Second, by observing the taint graph over time, we can learn about normal patterns/profiles in the taint graph. These profiles can then be used for anomaly detection, thus we can even detect attacks that we do not have specified policies for. Finally, the taint graph gives us the causal relationship which allows us to conduct diagnosis and analysis of malware behavior. Given a detection point or a malicious action, we can trace back to see how it happened, where the malware came from, and how it was installed, and we can also trace forward to see the subsequent actions that the malware has performed.

Note that even though information flow tracking has been proposed previously for intrusion analysis [21], these previous approaches such as Backtracker [21] suffer from several drawbacks: first, they are at process level which is often too coarse grained for malware detection and analysis; second, they do not apply to kernel attacks such as rootkit because they monitor program execution through system calls and do not monitor operating system behavior. By doing whole-system fine grained taint tracking, our method provides much higher accuracy than previous work and we can handle kernel attacks as well such as rootkit.

To examine this approach, we further design and implement a prototype called TaintQemu to analyze and detect a wide spectrum of malware in Windows, because the majority of malware aims for Windows platforms. Hence, the following discussion assumes Windows system, in particular, Windows 2000 and above versions, although the fundamental technique can be adapted to the other operating systems. Through extensive experiments, we demonstrate that taint graph based analysis correctly characterizes malware behavior and also provides unique insights, which cannot be obtained from the traditional analysis approaches. By applying taint graph

based policies, we can successfully detect a wide spectrum of malware. In performance evaluation, we observe that fine-grained taint analysis suffers up to 30 times performance overhead. Since TaintQemu is used for analysis and off-line detection, the performance overhead is not a crucial problem.

Contributions. In summary, this paper makes the following contributions. We propose a novel approach for malware detection and analysis: whole-system fine-grained taint tracking. We have designed and implemented a system TaintQemu to demonstrate that our approach provides a unified framework and is effective against a wide spectrum of different malware. Our approach has the following salient features: (1) it does not rely on signatures and thus can detect new attacks, (2) it prevents malware from evading detection by avoiding previously identified watch points, (3) it resides completely out of the victim system, and thus resists being disrupted by malware, and (4) it relies on hardware-level information and minimal software-level information, and hence is resilient evasion attacks by providing misleading information.

2 Overview

Whole-system fine-grained taint analysis is a novel approach for malware analysis and detection. Instead of performing analysis and detection within the victim system, we run the victim system as a virtual machine on top of this analysis and detection environment. This architecture provides an excellent isolation so that analysis and detection will not be disrupted by the malware inside the virtual machine. Figure 1 depicts our system architecture.

The Taint-Tracking Engine monitors system execution and tracks how tainted data propagates and generates taint graphs. The taint graphs will then be used for profiling, malware detection and analysis.

In the Taint-Tracking Engine, we selectively mark the data from the virtual devices, such as keyboard, network interface, and disk, as tainted, and observe the tainted data propagates in the system. During taint propagation, we can know which instruction access the tainted data in which register or memory location, and other hardware-level states. With the knowledge of memory map from the victim system, we can find out which process and module an instruction comes from. Combining raw events with memory map, we can generate taint graphs with different levels of granularity.

A taint graph presents process and module level information and the dependency and chronological relations between nodes. Therefore, from a taint graph, analyzers can easily understand the system behavior related to the tainted data.

A PolicyDB contains policies that specify properties which should be satisfied by taint graphs. Violations of these policies will be detected by the Malware Detection Engine which detects the malicious behavior of a malware. The policies in the PolicyDB can be manually specified by identifying intrinsic properties of different classes of malware, or be automatically generated by the Profiling Engine which builds profiles of normal executions during a training period.

Given an event of interest such as the detection of a malicious behavior, the Malware Analysis Engine performs backwards and forward reachability analysis to identify where the malware came from and how it was installed and the actions of the malware before and after the detection step.

3 Design and Implementation of Whole-system Fine-grained Taint Tracking

3.1 System Overview

The whole-system taint analysis requires that a certain kind of data from an I/O device be marked, and whole-system execution be monitored to find out what code regions have accessed and propagated this data or the data derived from it. Achieving this goal necessitates a whole-system emulator.

Therefore, we design and implement our system, TaintQemu, based on QEMU [28, 2], a generic and open source processor emulator which achieves a good emulation speed by using dynamic translation. The dynamic translation performs a runtime conversions of the target CPU instructions into the host instruction set, and the resulting binary code is stored in a translation cache so that it can be reused later. Hence, the emulation speed is improved drastically, compared to the previous emulation approaches (e.g. Bochs [5]).

For the case where the emulated CPU is the same to the host CPU, the QEMU Accelerator Module (KQEMU) is available to run most of the target application code directly on the host processor to achieve near native performance, as the other hardware-level virtual machine monitors (e.g. Xen [1], and VMWare Workstation and Server products) do. Currently, QEMU statically chooses to run in two modes: emulated mode (with KQEMU disabled), and virtualized mode (with KQEMU enabled). We have made a small modification on QEMU, such that it can switch between these two modes at runtime. This feature provides flexibility that QEMU can run in virtualized mode for the best of performance, and switch to emulated mode when performing analysis and detection tasks.

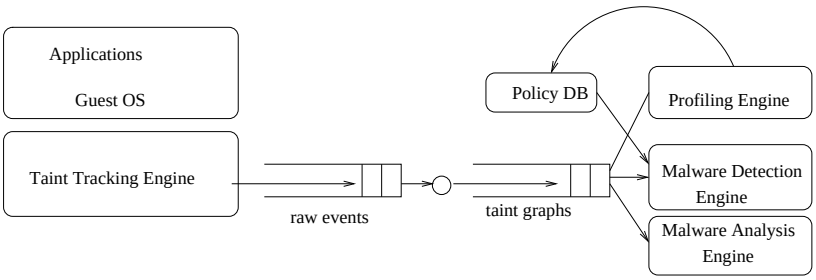


Figure 1: System Overview

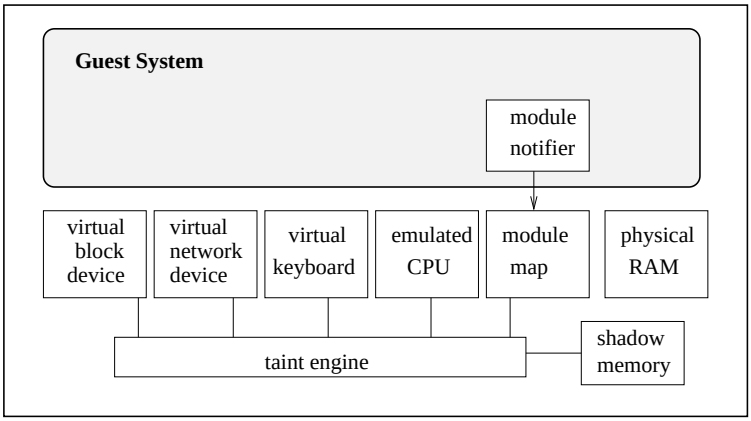


Figure 2: Architecture of TaintQemu

The architecture of TaintQemu is shown in Figure 2. We will describe its components respectively in the remainder of this section.

3.2 Shadow Memory

We use a shadow memory to indicate the status of each byte of physical memory and CPU's general-purpose registers. Each tainted byte is associated with a small data structure where its data source and some other book keeping information are stored. The shadow memory is organized in a page-table-like structure to ensure that it uses as little memory as possible in practice.

For simplicity, in the implementation, debug registers, control registers, SIMD(e.g. MMX and SSE) registers, and flags are not considered, and adding the necessary tracking for these registers would be quite straightforward.

Page swapping makes taint tracking more complicated, considering that tainted data could be swapped out to the disk and swapped in later. To accommodate page swapping, we modify the virtual disk to store an on-disk data structure mapping the location of tainted data. If data is re-read then the memory it is placed in is marked appropriately.

3.3 Taint Sources

In the current design and implementation, we consider to taint input from keyboard, network interface, and hard disk. Although tainting a high-level abstract data object (e.g. the output of an function call, or a data structure in a specific application or the OS kernel) is applicable, we do not explore this area in this paper, which certainly deserves investigation in future work.

Keystroke tainting simply taints bytes as they are read from the virtual keyboard. We only taint character keys (such as letters and numbers), which are the interest of malicious programs like keyloggers and password thieves. Moreover, since typing keyboard is a user interactive behavior, we let user to determine when to taint character keystrokes. We define a hot-key (i.e. ALT+CTRL+K), and before entering sensitive information, such as password and credit card numbers, the user activates the keystroke tainting by pressing the hot-key. Pressing the hot-key again will de-activate the keystroke tainting. [1] We assign a unique id for the tainted keystrokes during one activation. Therefore, there will be one taint graph generated for the keystrokes in one

[1] TaintQemu does not forward this hot-key to the virtual machine, so malware in the virtual machine cannot evade the detection by checking the hot-key.

activation.

Tainting incoming network packets from the network card is slightly more complicated. We have different tainting strategies for ICMP, and TCP/UDP packets. We uniformly taint all ICMP packets, so that there is one taint graph for all ICMP packets. For a UDP or TCP packet, we taint the whole packet, and derives its unique id from its source and destination IP addresses and port numbers. So a taint graph is generated for one TCP connection or one UDP virtual connection.

Tainting disk input is performed at the granularity of sector. We treat a sector as one unit and assign the sector number as unique id. As a result, a taint graph is generated for one sector.

3.4 Taint Propagation

After a data source is tainted, we need to track each instruction that manipulates data in order to determine whether the result is tainted. In QEMU, target CPU instruction is split into a few micro operations, which fall into three categories: data movement operations, arithmetic operations, and those that do neither. We instrument data movement operations and arithmetic operations. For data movement operations, the destination will be tainted if and only if the source is tainted. For arithmetic operations, the result will be tainted if and only if any byte of the operands is tainted. This is similar to the taint tracking approaches proposed recently [9, 16, 25]. There are a number of situations where the above basic propagation policy fails to taint correct information, which has also been discussed in [9].

Constant function Some instructions or instruction combinations always produce the same results. A quite common example is "`xor eax, eax`" in many IA-32 programs. After executing this instruction, the value of `eax` is always zero, regardless of its original value. If the input of this kind of instructions is tainted, the output should not be tainted. However, the basic propagation policy will still taint the output. TaintQemu recognizes these special cases such as `xor eax, eax` and `sub eax, eax` and sets the result location to be untainted. Note that there can be more general cases of constant functions where a sequence of instructions computes a constance function. We do not handle these more general cases. However, such cases are fairly rare.

Table lookup The second situation is table lookup. Sometimes a tainted value is used as index to read an entry in a non-tainted memory region. The basic propagation policy will check the non-tainted memory region and treat the output as non-tainted. This situation arises routinely for keystrokes. In Windows 2000 and above versions, a keystroke from the hardware is converted from scancode into virtual-key code, and then from a virtual-key code to unicode. These two conversion involve table lookup. To handle this special case, we augment the propagation policy with the following rule: if any byte of the index is tainted, then the result is tainted. However, accommodating table lookup may cause a great deal of innocent data regions being tainted. We have observed this phenomenon in our experiments. To retrain this "over-tainting" behavior, we associate a table-lookup depth value with each tainted byte. The initial value is 0. Whenever a tainted byte is derived from a table lookup, we increase the table-lookup depth of the tainted index by one, and assign it to the destination tainted data. In this way, we keep record of how many table lookup operations a tainted byte is derived from. By defining the maximum of table-lookup depth a tainted byte can have, we limit the maximal number of table lookup operations, and thus reduce false positives. We set the maximum of table-lookup depth as 2 in TaintQemu, and did not observe noticeable false positives.

Control flow evasion The third situation is control flow evasion. By comparing a tainted input with a sequence of clean data, the output data can be derived from the input from the comparison results, without being tainted. The following example illustrates this situation.

```
char copy(char x)
{
  char y=0;
  for(char i=128; i>=0; i>>=1)
    if(x>=i) y+=i, x-=i;
  return y;
}
```

The above function `copy` copies a character `x` to `y` without propagating the taint. Within a loop, the input `x` is compared with a value in `i` derived from a constant for 8 times, and the output `y` finally obtains the value of `x` with the knowledge of the eight comparison results. TaintQemu does not taint comparison flags and the output of instructions that follow a control flow decision. Even if comparison flags are tainted, it is difficult, if not impossible, to correctly identify the output of which instructions should be tainted, given the fact that code optimization techniques may reorder the instructions and put irrelevant instructions after branches.

This situation is fairly rare in regular code, but does exist in keystroke propagation in Windows 2000 and above versions. In our experiments on Windows XP, the unicode characters derived from keystrokes are not tainted as expected. After extensive kernel code tracing under SoftICE environment with Windows XP Service Pack 2 retail symbols, we have identified that

taint tracking stops at keystroke unicode conversion in an internal function `_xxxInternalToUnicode` in `win32k.sys`. Chow et al. faced the same obstacle in their implementation of TaintBochs [9], but did not have a solution. The translation of scancode into unicode involves a loop, illustrating such a control-flow evasion. We workaround this problem by patching an instruction in another internal kernel function `_xxxTranslateMessage` in `win32k.sys`. This instruction will write a translated unicode character to its destination register EAX. We instrument this instruction by re-tainting EAX when the corresponding scancode is tainted.

Note that malicious code cannot exploit the control flow evasion to evade detection, because it has to access the tainted data in the first place to launch the control flow evasion, and such an access will be caught by our detection.

Propagating to I/O devices Taints may also propagate to I/O devices, such as disk and network interface. Propagating to disk may result from a tainted page being swapped out to the page file, or tainted data being written to files. We maintain an on-disk data structure mapping the location of tainted data. If data is re-read then the memory it is placed in is marked appropriately. When generating taint graphs, we are only interested in tainted data being written to files, because tainted pages being swapped out is a normal system behavior. Thus, we need to distinguish these two situations. In current implementation, we configure the guest system to use page file on a separate virtual disk to facilitate distinguishing these two situations. When tainted data is written to a file, we can tell from the sector number that while file is being written to, by using disk analysis tools.

Taints may propagate to network interface, when tainted data is sent out. In the virtual network interface, we check if an outgoing packet is tainted.

3.5 Code Origin Resolution

When observing that an instruction is accessing tainted data, we need to know where this instruction comes from. The majority of instructions come from binary files on the disk, each of which is mapped into a code region in the memory. Such a code region is usually called *module*, in Windows and UNIX-like operating systems. In particular, device drivers with the extension of `sys`, shared libraries with the extension of `dll`, and executables with the extension of `exe`, are modules once loaded into memory space. In unusual situations, instructions may also be dynamically generated and executed on the heap.

Thus we need to maintain a virtual memory map, with the knowledge of what virtual memory location of which process each module is loaded into. Note that while a kernel module once loaded is shared by all processes, a user module is loaded into only one process memory space. Hence, this memory map consists of one global module list for kernel modules and a number of user module lists for all running processes. Maintaining this memory map requires the information from the guest operating system.

To obtain necessary information, we have implemented a kernel module called *Module Notifier*, and insert it into the guest system, to push down the updated memory map information to the underneath TaintQemu. Module Notifier obtains the information by calling two kernel APIs `PsSetCreateProcessNotifyRoutine` and `PsSetLoadImageNotifyRoutine` to register two callback functions. The first callback function tells which process is being created or deleted. The second callback function is called whenever a new module is loaded, with the information which part of virtual memory of which process it is loaded in. In addition, Module Notifier obtains the value of CR3 for each process. As CR3 contains the physical address of page table of current process, CR3 is unique for each process. All the above information is submitted to the underneath TaintQemu through a predefined I/O port. By receiving the memory map information from the guest system, TaintQemu can reconstruct a map of all modules currently loaded into the guest system. Then when observing an instruction is accessing tainted data, we can search the map with CS, EIP, and CR3, and find out which module this instruction resides in.

We rely on the following security mechanisms to guarantee the integrity of modules loaded into memory. In modern operating systems, file integrity checker is commonly available as a security feature, such as Windows File Protection (WFP) [39] for Windows and Tripwire [20] for UNIX family. This feature assures the integrity of critical system files. After the module is loaded into memory, the virtual machine monitor can further prevent this module from being tampered, by leveraging the virtual machine introspection technique [13].

The virtual memory map is an essential component in our system, and the only one obtained from the target system. Therefore, we have to make sure the authenticity of the memory map information. To disrupt our detection, a malicious program may inject fake information to the predefined I/O port. In order to ensure the authenticity of the messages from the target system, we check the program counters of the instructions, and only allow those from Module Notifier to send information to the predefined I/O port.

4 Taint-Graph Based Detection and Analysis

4.1 Taint Graph

Taint propagation forms dependencies among instructions and hardware inputs and outputs (e.g., memory, keyboard input, network interface input/output, disk). For example, if instruction B reads a tainted data written by instruction A, then instruction B depends on instruction A. Thus, the chain of instructions that operated on and propagated tainted data form a graph, where the nodes represent the instructions and hardware inputs and outputs, and an edge between two nodes indicate an immediate taint propagation relationship. We call this graph the *taint graph*. The taint graph contains all the information about how taint propagates during the execution which provides a foundation for us to perform detection, profiling, and analysis.

However, such a fine-grained taint graph is often enormous—a short-duration execution could result in a taint graph of gigabyte size. Thus, we provide different options to reduce the taint graph by abstracting away certain detailed information and representing the taint graph at different levels of granularity. For example, instead of keeping information about each instruction propagated taint, we simply keep the information about the module that the instruction belongs to (including the process name and the module name) and collapse consecutive nodes with the same process and module name. In this case, the reduced taint graph represents the dependencies among different modules (instead of instructions) which often provides sufficient level of granularity for malware detection and defense. Figure 3 demonstrates a small part of a taint graph at the module level. However, a taint graph at the module level can still be very large, because there can be many system kernel and user modules and they could appear repeatedly in the graph.

As the focus of our analysis is the untrusted code [2], we can simply treat all trusted system kernel modules as one pseudo module "SYS", and all trusted system user modules as another pseudo module "USR". This step greatly reduces the complexity of the taint graph even further, without losing the dependency between the untrusted code and system code. We demonstrate the effectiveness of this strategy using an example of incoming FTP traffic, shown in Figure 3.

[2] By untrusted code, we generally mean the code that needs to be investigated, which can be the programs explicitly installed, or those surreptitiously penetrating the system

4.2 Taint-Graph Based Policy for Malware Detection

Given a taint graph, we can define various security properties or predicates on the graph such that the violation of the properties or predicates will indicate a malicious or suspicious behavior. We call these security properties and predicates *taint-graph based policy*, which will be used for malware detection. The taint-graph based policy can be specified in different logical forms. For example, it can be a path property, i.e., a predicate over all paths in the graph. It can also be a logical form over different taint-graphs which enables us to correlate different taint-graphs for more sophisticated attacks and detection.

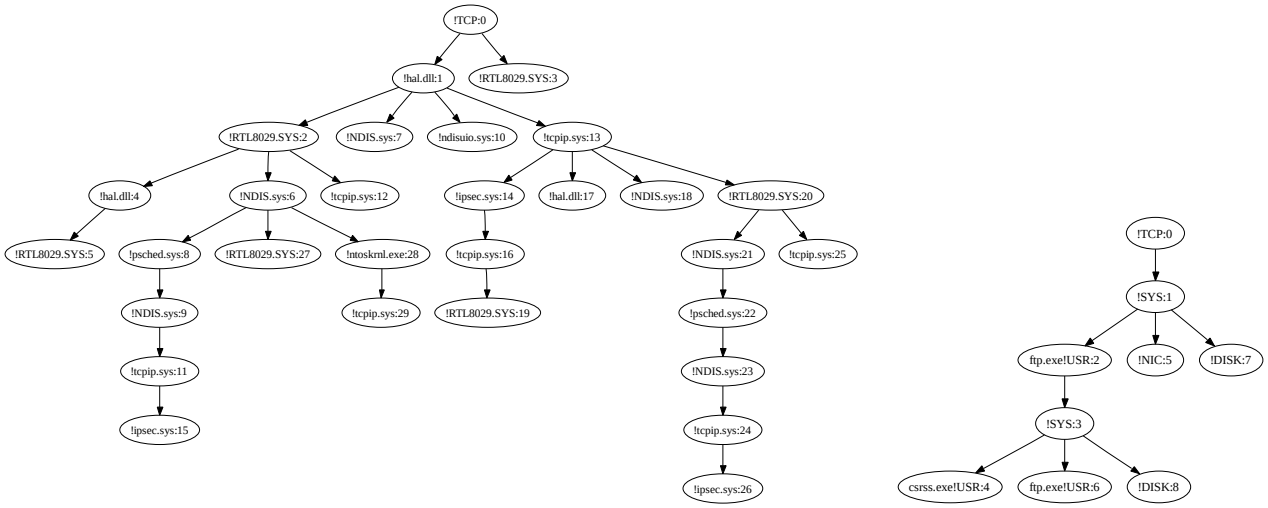
The taint-graph based policy is the heart of the malware detection using taint tracking. Given the policy, the detection is simply to evaluate the predicates to check for any violations. Once we detect a malicious behavior, we can then perform further analysis using the taint graph as described in section 4.3.

There are two ways to generate the policy. First, by analyzing the intrinsic properties of malware, we can manually specify policies that will capture different classes for malware such as keyloggers, back-doors, password thieves, etc. Second, by analyzing the common properties of normal taint graph (i.e., taint graphs that do not contain malicious behavior execution), we can build profiles of normal behavior of taint graphs for different applications, and use the deviation from the profile to detect malware. We provide more details for these two types of policy generation below.

Statically Defining Policies Different classes of malware have different intrinsic properties or invariants which we can leverage to specify policies for accurate detection. We give a few examples here to illustrate how we specify policies for accurate detection of different classes of malware. In Section 5, we demonstrate with real malware samples that these policies are truly effective in detecting a variety of different malware and cause no known false positives.

First, incoming TCP/UDP traffic or keyboard input is always consumed by user-level applications. The data from hardware goes through the OS kernel and reaches the application directly. Thus, all taint graphs with the root as TCP or UDP or KBD should always follow such a pattern. Any violations are highly suspicious. For example, kernel-mode keyloggers tend to intercept keystrokes in the kernel, and break this pattern. Similarly, kernel-mode packet sniffers and backdoors intercept incoming network traffic, and break this pattern.

Second, ICMP is designed for network testing and diagnosis purpose, and usually only the operation system and trusted testing tools (e.g. `ping.exe`) use it. Un-



(a) Small part of original graph (b) Compressed graph

Figure 3: **Taint Graph Compression.** The original graph contains 1624 nodes, and (a) only shows the first 30 nodes. After compression, the reduced graph only contains 9 nodes, shown in (b).

trusted code that accesses incoming ICMP traffic is often suspicious. Most of packet sniffers tend to capture all traffic, including ICMP packets, and thus will access it. Some stealth backdoors, such as SAdoor [34], uay [36], and deepdoor [33], achieves stealthiness by sniffing the incoming traffic, and thus also access it. ICMP based backdoor is another kind of stealth backdoor that uses ICMP messages as a vehicle to carry backdoor communication, and therefore will access ICMP traffic. Thus, by simply stating a policy that untrusted code should not be on a path from an ICMP input, we can detect ICMP-based backdoor, packet sniffers, and sniffing-based backdoor.

Third, we consider malware that tries to steal users' sensitive file and send it over the network. In this case, users' sensitive file will be tainted inputs, the policy will simply be specified as untrusted code should not be on a path from a sensitive file to the network in a taint graph. In Section 5, we will demonstrate how we use such a policy to detect when a malware tried to send out a password file.

Automatically Generating Policies through Profiling To detect new classes of malware and detect application-specific attacks, we also enable the approach of anomaly detection on taint graph. For an application, through a training period with no malware installed, we collect taint graphs and build profiles which holds across different taint graphs. These profiles can then be used to detect malicious code when untrusted code deviates from the profiles. A profile can include a lists of modules and hardware nodes that access the tainted data, and probably the dependency and chronological relations between them. Currently, we only consider the basic profile that includes only a list of modules and hardware nodes. For example, for an application, we can build a profile to see which modules normally would access keyboard input for the application. We can then use this profile to detect user-level keylogger when the keylogger (the untrusted code) is trying to access keyboard inputs for this application. For a detailed example, see Section 5.

4.3 Taint-Graph Based Analysis

A taint graph records data source, a list of modules and hardware nodes that access the data, and the dependency and chronological order between nodes. Thus, we can diagnose and analyze the system behavior via taint graphs. Given any event of interest such as a node causing the detection of a malicious behavior (called the detection step), we can perform backwards and forwards reachability analysis on the taint graph. The backwards reachability analysis can lead us to where the malware has been downloaded and installed and reveal what other actions have been performed by the malware before the detection step; the forward reachability analysis can tell us what other actions and consequences has this malicious behavior led to. Such analysis capability can be instrumental for system administrators and analyzers to understand the behavior and consequences of malware. For a simple example, we can see from Figure 3(b) that a TCP connection is received by `ftp.exe` program and the data in that connection is stored into disk, and some information is sent to `csrss.exe` for console display. This

graph correctly reflects the real behavior of `ftp.exe` when handling incoming TCP traffic. In Section 5.2, we will further examine the system behavior when malware is involved in the taint graph.

5 Evaluation

In this section we present an experimental evaluation of our TaintQemu system. Our evaluation consists of three parts. First, we test the effectiveness of taint graph based detection against some common malware instances. Second, we evaluate the effectiveness of taint graph based analysis. The third part evaluates the performance overhead. Note that since TaintQemu is used for analysis and off-line detection for now, the performance is not a problem.

5.1 Malware Detection

```
int Policy1(vector<Node> &graph)
{
  if(graph[0].mod in {"UDP", "TCP", "KBD"}){
    if(graph[1].mod!="SYS") return error;
    if(graph[2].mod!="USR") return error;
  }
  return ok;
}
int Policy2(vector<Node> &graph)
{
  if(graph[0].mod=="ICMP"}){
    for(int i=1; i<graph.size(); i++)
     if(graph[i].mod not in {"SYS", "USR"})
       return error;
  }
  return ok;
}
int Policy3(vector<Node> &graph)
{
  for(int i=3; i<graph.size(); i++) {
    if(graph[i].mod not in
      Profile(graph[2].proc, graph[0].mod))
    return error;
  }
  return ok;
}
```

Figure 4: Sample Policies

5.1.1 Sample Malware Instances

To extensively evaluate the effectiveness of TaintQemu, we choose to run a wide spectrum of malware, including keyloggers, password thieves, packet sniffers, stealth backdoor, and rootkits, as described below.

Keyloggers capture keystrokes and attempt to extract sensitive information from them. They can reside in both user and kernel space. `klog` [23] is a kernel-mode keylogger implemented as a filter driver. It only functions properly for Windows NT, and we have modified it to enable it run properly in Windows 2000 and XP. `Vanquish` [37] is a user-mode keylogger, which is automatically loaded into each process that is receiving keyboard input.

Password thieves steal passwords and potentially other sensitive information, which are usually derived from keyboard input. `GINA Spy` [14] replaces Windows `msgina.dll` with a malicious `mscad.dll`, which steals Windows username and password in `Winlogon.exe`. `BHO Spy` is a synthetic malicious BHO (i.e Browser Helper Object) written by one of the authors to steal passwords that the user enters into the web documents within Internet Explorer. `Password Logger` [26] is a standalone application running in the background. It scans for password fields [3] in all GUI applications, and saves the content into a file.

Packet sniffers eavesdrop incoming and possibly outgoing network traffic to capture sensitive information. They can reside in both user and kernel space, interposing every possible point in the network stack. `IP sniffer` [18] demonstrates a driverless userland implementation of IP sniffer using raw socket.

Stealth backdoors attempt to communicate with remote attackers, but at the same time evade detection by the host security mechanisms such as personal firewalls. To achieve the stealthiness, some backdoors like `Back Orifice` [4] inject code into trusted applications that are allowed to access network by the system policy. Some others avoid opening ports. ICMP backdoors utilize non-port-based ICMP protocol as a vehicle to convey backdoor communication. `ICMP_backdoor` [17] demonstrates such an ICMP backdoor. `UAY backdoor` [36] is a portless kernel-mode backdoor. It hooks on the network stack and intercepts all incoming TCP packets with a specified destination port number (the default port is 9929). After receiving a command from a remote attacker, it executes the command in the context of a system kernel thread.

Rootkits intend to conceal the presence of malicious activities, such as files, registry entries, processes, modules, and other system states, which helps an intruder maintain access to a system without the user's knowledge. `CFSD` [8] is a kernel-mode rootkit that hides directories and files. Beside a userland keylogger, `Vanquish` [37] is also a userland rootkit that hides directories, files, and registry entries.

[3] A password field is an input window that characters in it are masked.

Name	Description	Detected Module	POL1	POL2	POL3
klog	kernel-mode keylogger	`klog.sys`	✔		
Vanquish	keylogger and userland rootkit	`vanquish.dll`			✔
GINA Spy	Windows account password thief	`mscad.dll`			✔
BHO Spy	BHO stealing password in IE	`BHOspy.dll`			✔
Password logger	password thief	`PasswordLogger.exe`			✔
uay	kernel-mode backdoor	`uay.sys`	✔	✔	
ICMP_backdoor	backdoor communicating via ICMP	`ntkrnl.exe`		✔	
IP sniffer	userland packet sniffer using raw socket	`iptools.exe`		✔	✔
CFSD	kernel-mode rootkit	`cfsd.sys`			✔

Table 1: Detection results against sample malware instances.

5.1.2 Detection Results

We first run a clean system with Window XP Professional SP2 installed on top of TaintQemu, and perform some normal operations on it. These operations include entering password for Windows authentication, enumerating files and directories in console, using IE to visit website and check email, using ftp to download files, and some others. In these period, we build profiles for those applications that are involved in taint graphs. Then we install the above malware samples, and perform the same operations. Then we use the sample policies in Figure 4 to check the taint graphs.

The experiment shows that the policies have successfully detected all malware samples. `klog` and `uay` attempt to intercept the path that the data from hardware passes through kernel and reaches the application in user space, and thus violate Policy 1. `uay`, `IP sniffer`, and `ICMP_backdoor` violate Policy 2, because they access the incoming ICMP traffic. Note that although `uay` does not use ICMP, but it has to check the ICMP header to determine if it wants the packet. `IP sniffer` also violates Policy 3, because it has accessed incoming TCP traffic for IE and ftp, violating their TCP profiles. `Vanquish` violates Policy 3, because it accesses the keystrokes belonging to other applications. We have observed that for those applications receiving keyboard input, their KBD profiles are violated. `GINA Spy` and `Password Logger` steal the password in `Winlogon.exe`, and thus violate the KBD profile of `Winlogn.exe`. `BHO Spy` steals the password in IE, violating the KBD profile of `IEXPLORE.exe`. `Vanquish` and `CFSD` violate the DISK profile of `cmd.exe`. When we enumerate files and directories in `cmd.exe`, they check the filesystem meta data from disk, and tamper it to hide files and directories. In summary, we list the detection results in Table 1.

5.1.3 False Positive Analysis

We also evaluated the false positive rate of our detection policies by running the detection system on a clean system over an extended period of time while we run different applications. Our detection policies did not trigger any false positives.

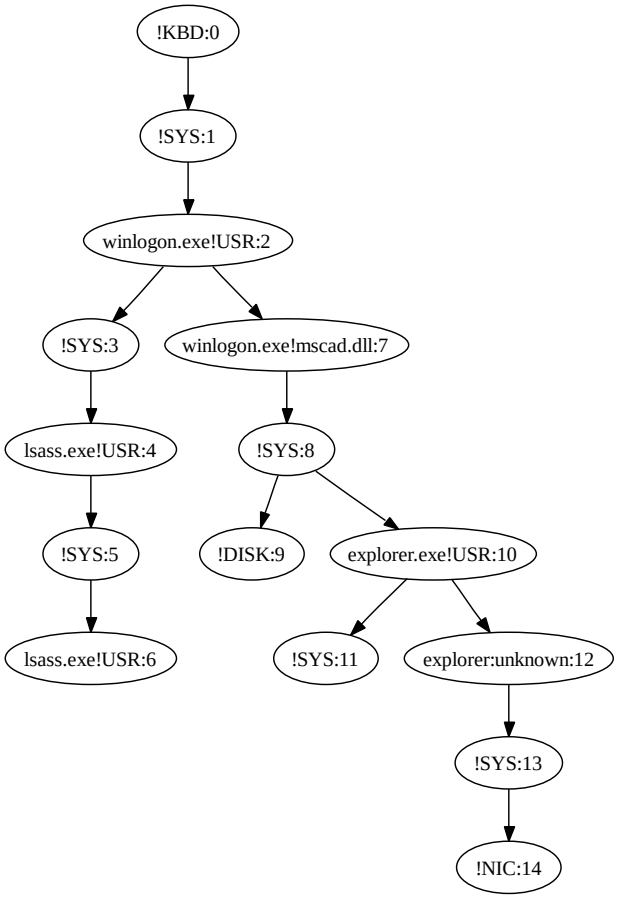


Figure 5: Taint Graph for Analysis

5.2 Malware Analysis

To evaluate the effectiveness of our malware analysis using taint graphs, we setup the following attack scenario. We installed GINA spy [14] to steal user's Windows password and store it into a file, and Back Orifice 2000 (bo2k) [4] as a backdoor to send the file to the remote attacker. We tainted the keystroke input when typing password for Windows authentication. We kept the system running for a while (about 1 hour). Then we behaved as a remote attacker to retrieve the file containing password from another machine, using the bo2k client. The taint graph recording the whole procedure is shown in Figure 5.

We have several observations from the taint graph. First, there are two untrusted modules: `mscad.dll`, which resides in the process `winlogon.exe`, and a module from unknown source loaded into the process `explorer.exe`. The first one is the module for GINA spy, and the second is the code injected by bo2k to the heap of `explorer.exe`.

Second, when we trace back and forward the taint graph for `mscad.dll`, we can see that `mscad.dll` obtained the password from winlogon and then stored it into disk. In more detail, we actually know the sector number containing the password. Therefore, using a disk analyzing tool, we can eventually find out which file the password is stored in.

Third, when we trace back and forward the taint graph for "`unknown`" injected by bo2k, we can see that "`unknown`" read the password cached in the kernel buffer and then sent it out of network. Certainly, we can analyze the packet header of the tainted outgoing packet and obtain the destination IP address and port number. Thus we can know where the remote attacker is.

Note that previous information-flow tracking methods such as Backtracker [21] cannot provide such analysis information because it was too coarse-grained (e.g., only at process level).

5.3 Performance

We measure TaintQemu's performance using a benchmark tool called Nbench [24]. Nbench measures performance with respect to CPU, main memory and disk. For CPU, it measures integer and floating point operations speed (in MOP/s); for main memory, it measures the throughputs of random and serial access (in MB/s); for disk, it measures the read and write speeds (in MB/s). We run Nbench in four configurations: native execution, full virtualization, full emulation, and taint analysis mode. Full virtualization refers to TaintQemu with support of KQEMU, running most of target instructions directly on the host system. Full emulation refers to TaintQemu with KQEMU disabled, emulating all target instructions. The functionality of data flow probing is not included in this configuration. In taint analysis mode, the system emulates all target instructions, propagates tainted data, and generate taint graphs. The host system is a Dell workstation with two 3.2 GHz Pentium 4 CPUs and 1GB of RAM, running Windows XP Professional with Service Pack 2. We allocate 256M of RAM for the guest system, which also runs Windows XP with SP2.

Figure 6 shows the performance results. We normalize the results using the performance of native execution as baseline. The first bar gives the performance of full virtualization, showing that its CPU integer and floating point speed and serial memory access speed are comparable with those of native execution. Due to the overhead of emulating disk device, its disk read and write speeds drop 10-17%. The memory random access speed drops the most, probably because of the overhead of software MMU (Memory Management Unit) implementation in QEMU. The second bar shows the performance of full emulation. Full emulation incurs 7 and 10 times slowdown for integer and floating point operations respectively, coinciding the result presented in [2]. Its disk read and write speeds further drop a little, and its memory access speeds (both random and serial) drop to 20% of the native execution speeds.

The final bar gives the performance of taint analysis. Understandably, due to the cost of taint tracking, taint analysis suffers 30 times slowdown for CPU integer operations, and 20 times slowdown for memory random and serial accesses.

6 Discussion

Application Scenarios The current implementation of whole-system fine-grained taint tracking incurs considerable execution overhead. Thus, we envision that our system is used for off-line detection and analysis for now. Several application scenarios are applicable. First, antivirus companies and analyzers can use it as an analysis environment for evaluating and testing malware or untrusted code in general. Second, the end system runs as usual most of time and its system image is loaded periodically into our system to perform malware detection, in a manner similar to periodical virus scanning. Third, we can leverage the technique of virtual-machine logging and replay [12]. The end system is loaded in virtual machine and runs in fully virtualized mode, with necessary events being logged. At a later time, the virtual machine switches to taint-tracking mode and replays the execution for malware detection and analysis.

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

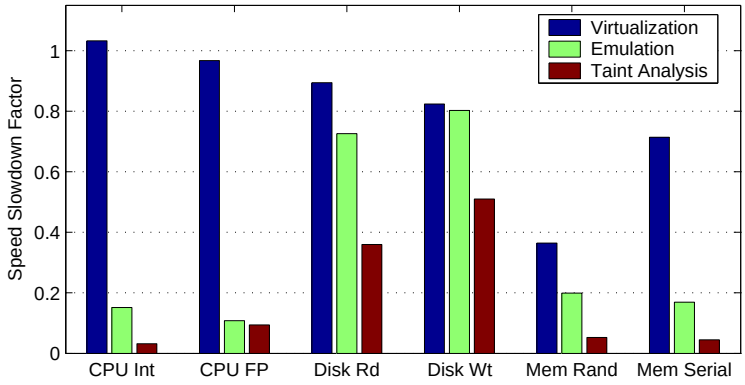


Figure 6: Speed Slowdown of TaintQemu

switches between virtualized and emulated execution, and emulation is only used when tainted data is being processed by CPU [16]. Exploring finer-grained hardware protection provided by ECC may further improve the performance greatly [29]. Therefore, with more efficient implementation of whole-system fine-grained taint tracking, our technique will be practical to be deployed for online malware detection and analysis.

7 Related Work

Other 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 [10] improves its resilience to polymorphic and metamorphic variants, the inherent limitation of 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 [7], 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 [38] checks auto-start extensibility points in the registry to determine surreptitious restart-surviving behaviors. VICE [6] and System Virginity Verifier [32] search for user- and kernel-level hooks in IAT (Imported Address Table), EAT (Exported Address Table), SSDT (System Service Descriptor Table), IDT (Interrupt Descriptor Table), IRP major function table, etc., which are usually used by rootkits and the other malware. Rootkit Revealer [30] and Blacklight [3] detect rootkits by comparing two views of the system: the upper-level view is derived from calling common APIs to enumerate key system elements, such as files, processes, registry keys, and so on, while the low-level view is obtained from system states in the kernel or from hardware if applicable.

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 does not rely on detection points, 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 cheated. Third, it is implemented completely outside of the victim system, and so strongly protected from being subverted.

Other virtual machine based approaches Virtual machines have been used to enhance security. Researchers have used virtual machines to detect intrusions [13, 19], and analyze intrusions [12, 21]. These approaches build security services within the virtual machine monitor, which is isolated from the system to be monitored. This architecture prevents the intrusions within the target system from disrupting the security services in the virtual machine monitor. The same technique can also be exploited for malicious intent. VMBR builds malicious functionalities within the virtual machine monitor and hides from the operating system running within a virtual machine [22], which calls for the detection to be performed at an even lower level.

Virtual machine based malware detection is a promising approach, due to the isolation provided by the virtual machine monitor. By directly observing hardware state and events and using this information to extrapolate

the software state in the target system, virtual machine based detection can achieve the functionality comparable to that of behavior based detection running in the target system. However, its power of detection still relies on the selection of detection points.

Our system is also based on virtual machine technique, and thus malware within the virtual machine cannot disrupt and hide from our detection. A distinct feature of our system is that it explores a novel virtual machine based architecture, in which the virtual machine monitor can sometimes switch to emulated mode and enable fine-grained (i.e. instruction-level) analysis and detection.

Other taint based approaches Dynamic taint analysis has been applied to solve and analyze other security related problems, such as worm detection and data lifetime analysis. Several approaches have been proposed to detect Internet worms. Newsome et.al. implemented a TaintCheck system by instrumenting Ring-3 instructions to detect and analyze worms, and automatically generate signatures [25]. Minos proposed a whole-system taint tracking to perform Biba-like data integrity check of control flow to detect exploits at runtime [11]. These systems 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 emulation with taint analysis to analyze how sensitive data are handled in operation systems and large programs [9]. The major analysis was conducted in Linux, with source code support of the kernel and the applications. Recently, Portokalidis et. al. proposed a QEMU-based taint tracking system for worm detection similar to the previous approaches [27]. Their work does not propagate taint to disk and is only for detecting overflow attacks as previous approaches. Our system is independently developed and provides a different machinery for malware detection. Note that malware detection and overflow attack detection require completely different methods.

Our system is the first work to use dynamic taint analysis for malware detection. Unlike [9], we cannot assume the availability of source code for the operating system and applications, since the majority of malware instances work on Windows systems. In contrast to the above systems, which can only be used as honeypot or absolutely for analysis purpose, due to the huge performance overhead of taint analysis, our system is practical to run on end systems by running in virtualized mode most of time and only switching to emulated mode to perform detection tasks infrequently.

8 Conclusion

As malware is becoming increasingly sophisticated and stealthy, existing techniques for malware detection and analysis become ineffective. In this paper, we have proposed *whole-system fine-grained taint analysis* for malware detection and analysis. We demonstrate that such whole-system fine-grained taint analysis can capture the intrinsic properties of many different classes of malware and thus offer effective methods for automatic malware detection and analysis. Our evaluation using a wide spectrum of real-world malware demonstrate that our system is effective in detecting many different classes of malware including keyloggers, backdoor, etc., and offer indispensable assistance to system administrators and analyzers for better understanding of the behavior and consequences of malware.

References

[1] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[2] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (April 2005).

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

[4] Back orifice 2000. `http://www.bo2k.com/`.

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

[6] BUTLER, J., AND HOGLUND, G. 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`.

[7] BUTLER, J., AND SPARKS, S. Shadow walker: Rasing the bar for windows rootkit detection. In *Phrack 63* (July 2005).

[8] Clandestine file system driver. `http://www.rootkit.com/vault/merlvingian/cfsd.zip`.

[9] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium* (August 2004).

[10] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 32–46.

[11] CRANDALL, J. R., AND CHONG, F. T. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture* (December 2004).

[12] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation* (December 2002).

[13] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium* (February 2003).

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

[15] Desktop search tools seen raising red flags. http://www.networkworld.com/news/2006/041706-desktop-security.html?nettx=041706netflash&code=nlnetflash30668.

[16] HO, A., FETTERMAN, M., CLARK, C., WATFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys'06* (April 2006).

[17] ICMP backdoor. http://www.heibai.net/down/show.php?id=5570.

[18] IP sniffer. http://erwan.l.free.fr/.

[19] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles* (October 2005).

[20] KIM, G. H., AND SPAFFORD, E. H. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security* (1994), pp. 18–29.

[21] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles* (2003), pp. 223–236.

[22] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H., AND LORCH, J. R. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy* (April 2006).

[23] Klog. http://www.rootkit.com/vault/Clandestiny/Klog%201.0.zip.

[24] Nbench. http://www.acnc.com/benchmarks/nbench.zip.

[25] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium* (February 2005).

[26] Password logger. http://iamaphex.net/downloads/PasswordLogger.zip.

[27] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys'06* (April 2006).

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

[29] QIN, F., LU, S., AND ZHOU, Y. 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* (Feb 2005).

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

[31] RUTKOWSKA, J. Detecting windows server compromises. In *Hivercon Security Conference* (November 2003). http://www.invisiblethings.org/papers/hivercon03_joanna.ppt.

[32] RUTKOWSKA, J. 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.

[33] RUTKOWSKA, J. Rootkit hunting vs. compromise detection. In *Black Hat Federal* (January 2006). http://www.invisiblethings.org/papers/rutkowska_bhfederal2006.ppt.

[34] SAdoor: A non-listening remote shell and execution server. http://cmn.listprojects.darklab.org/.

[35] Sony's drm rootkit: The real story. http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html.

[36] UAY backdoor. http://www.xfocus.net/tools/200602/uay_source.rar.

[37] Vanquish. https://www.rootkit.com/vault/xshadow/vanquish-0.2.1.zip.

[38] WANG, Y.-M., ROUSSEV, R., VERBOWSKI, C., JOHNSON, A., WU, M.-W., HUANG, Y., AND KUO, S.-Y. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for spyware management. In *Proceedings of the Large Installation System Administration Conference (LISA)* (November 2004).

[39] Description of the windows file protection feature. http://support.microsoft.com/?kbid=222193.