System-level Support for Intrusion Recovery

Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos
{abs204,rvn270,asia,herbertb}@few.vu.nl

Network Institute, VU University Amsterdam

Abstract. Recovering from attacks is hard and gets harder as the time between the initial infection and its detection increases. Which files did the attackers modify? Did any of user data depend on malicious inputs? Can I still trust my own documents or binaries? When malcode has been active for some time and its actions are mixed with those of benign applications, these questions are impossible to answer on current systems. In this paper, we describe DiskDuster, an attack analysis and recovery system capable of recovering from complicated attacks in a semi-automated manner. DiskDuster traces malcode at byte-level granularity both in memory and on disk in a modified version of QEMU. Using taint analysis, DiskDuster also tracks all bytes written by the malcode, to provide a detailed view on what (bytes in) files derive from malicious data. Next, it uses this information to remove malicious actions at recovery time.

Keywords: Attack recovery, dynamic taint analysis

1 Introduction

We describe DiskDuster, a semi-automated system to help recover from intrusions. Intrusions may result from remote attacks, open network shares, exploits (Conficker [22]), user-installed Trojans (some versions of Torpig [27]), etc. However it spreads, the malicious code may interfere in deep and involved ways with the system state and removing the infection and its effects is difficult. For instance, Torpig turns off anti-virus scanners, modifies data, steals confidential information, and downloads/installs more malware on the victim's computer. Other attacks destroy data, or encrypt files for ransom.

Our recovery procedure aims to return the system to a sane state, as existed just before the attack, while retaining as much of the recent user data as possible. We show that we can undo most of the effects of complicated attacks. As an example, we demonstrate the usefulness of our approach for drive-by-downloads that fetch and execute malware that subsequently modifies the registry, and infects other programs that, in turn, modify system state. And so on. See Figure 2 for a full description of our running example. We evaluate our solution with several real attacks on Windows.

Recovering from attacks Despite a plethora of defense mechanisms, attackers still manage to compromise computer systems. Sometimes they do so by corrupting memory and injecting a small amount of shellcode to download and install the real malware. Sometimes the users themselves install trojanized software. To make matters worse, the malware may be active for days before it is discovered.

Upon discovery of a compromised machine, one of the most challenging questions is: what did the malware do? Which files has it modified? Did the attackers change or corrupt my financial records? Can I still trust any of the files created after the compromise, or should I check each and every one manually? Did the initial attack spread to other programs? And, most importantly, can I *undo* the malicious actions and restore the system to a sane state, without losing my recent data?

Currently, the only sane state a system can revert to is the last known good backup. This leaves the question of what to do with the changes to the system that occurred since then. Ignoring them completely is safe, but often unacceptable—losing valuable data generally is. Accepting them blindly is easy, but not safe—modifications may be the result of the malware's actions. However, the alternative of sifting through each of the files (or even blocks) on disk one by one to see whether it can still be trusted may be too time-consuming. Thus, we developed DiskDuster to automate most of this process.

High-level overview Figure 1 illustrates DiskDuster's main flow of operations. The circled numbers in the text below correspond to the numbers in the figure. To minimize the performance impact, and to retain as much of information about the attack as possible, we decouple the analysis and recovery from the production machine. Thus, DiskDuster records the execution on the live production machine ① and replays it ② on a dedicated security server with additional security checks and recovery operations ③.

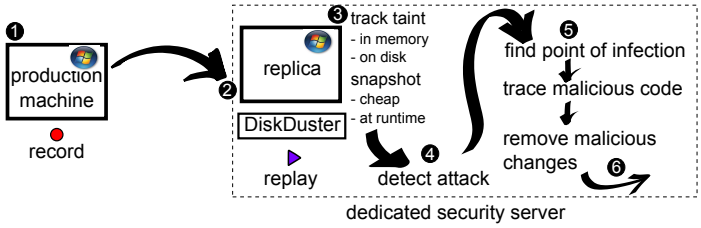


Fig. 1. Intrusion recovery in a decoupled security model.

To recover the user data after an infection, we assume the presence of at least one detection method ④. The nature of the detection method is not important. The prototype in this paper works with dynamic taint analysis (DTA) and AV scanning, but we can easily add system call analysis or other techniques.

As soon as DiskDuster detects an intrusion in the replay, the user shuts down the original machine, while the security server continues to replay the trace, using DTA to monitor all the malcode's actions ⑤, tainting all writes by the malcode to memory and disk as malicious. Taint propagates whenever the malicious bytes are read, copied, or used in ALU operations. If malicious bytes compromise other processes, DiskDuster traces those also. Finally, DiskDuster cleans up the system by replaying benign disk writes up to the moment of infection. For the time between the infection moment and the detection moment, DiskDuster classifies all disk writes as 'benign' (not affected by the attack), 'malicious' (written by a malicious process) and 'suspicious' (possibly affected by the attack). Only suspicious data requires manual intervention.

Contributions Most existing intrusion recovery approaches assume that the infection *cannot spread to the kernel* itself [15, 3, 14]—a very strong assumption that typically

does not hold in practice. Others provide very limited protection (e.g., modification, but not removal, of system state and a few system files only [20]), or require users to define trusted data and malware manually [3]. Finally, existing approaches are typically tied to specific operating systems (often Linux, to have access to source code) [11, 28, 14].

In contrast, DiskDuster operates at the level of the (virtual) hardware and the approach can be applied to any OS. Throughout this paper, we focus on Windows, as it is (still) easily the most popular attack target. In addition, DiskDuster protects both the kernel and user processes and handles modification and removal of any file.

Thus, the contribution of this paper is an intrusion analysis and recovery system on top of a hardware emulator that works with *unmodified* OSs and applications and protects both kernel and user processes against complicated attacks. Our goal is to recover user *data*, but the system helps to recover other files and folders also.

Moreover, where modern tainting systems typically detect or track an attack on a single process, DiskDuster tracks the attack and all related processes, as well as their spread throughout the system. For instance, we track all disk writes of the malicious code, and take appropriate action when a benign process reads such bytes. Likewise, we treat processes that are started by a malicious process as malicious also. The same is true for threads injected by malicious code in a benign program. We are not aware of other systems with the same comprehensive tracking of malicious activity.

Tracking infections requires tracking the actions and data generated by the attack. Specifically, we need to know where this data ends up and what actions and data depend on it. Where almost all state-of-the-art intrusion recovery solutions [14, 20] construct dependency graphs explicitly, DiskDuster tracks dependencies directly, by means of dynamic information flow tracking (taint analysis) and at byte-level granularity. Doing so is simpler and potentially weaker. But as it requires very little knowledge of the OS, it enables us to (a) support different OSs, and (b) handle kernel infections also. Moreover, we will see that the way DiskDuster handles implicit flows is very simple and yet very powerful. It allows it to limit taint tracking to explicit flows during analysis, while not losing even a byte of implicitly modified data (although overtainting may well occur).

Clearly, recovery cannot be complete if the attack had side effects beyond this system. For instance, if the malware sent spam, or leaked information to an external party, there is no way to undo this. We do revert changes on the file systems. We think this is sufficient for cleaning up infections locally. Even if some (memory-resident) attacks do not themselves leave any presence on disk, this is not a problem for DiskDuster. As long as it can detect the attack (e.g., using taint analysis), it will remove all disk writes that the malware influenced, while the malware itself will disappear after the reboot.

2 Threat model and assumptions

The ideal intrusion recovery system, upon detecting an attack, removes all harmful actions related to the attack automatically, leaving only changes to the system unaffected by the attack. Fundamentally, this is not possible—at least not in the general case. For instance, after an attack deletes the AV binary, a legitimate user may write a memo: “No AV scanner present”. Automated recovery may restore the AV scanner, but cannot spot the relation with the memo, resulting in inconsistencies (see also Section 5).

A nasty attack

The effects of real-world attacks like Torpig and Conficker have been complex and devastating. In this paper, we combine in our running example the effects of these and a number of others to create a complicated attack:

1. A drive-by-download infects the browser.
2. The attack immediately migrates to another running process on the same machine—infecting this process also. The migration complicates the tracking, since the second process did not connect to a malicious website.
3. The attack deletes the antivirus program.
4. Next, the shellcode in the second process downloads and executes the real malware and adds a registry key to make itself persistent across reboots.
5. Later, the malware encrypts the 'Documents' folder on disk, for ransom purposes, while deleting itself to prevent security experts from reverse engineering it.

Goal: to clean up the system and remove all traces of the attack.

Fig. 2. Attack scenario used as a running example.

In practice, however, (semi-)automated recovery can be a powerful tool in post-hoc sanitization. By tracing what data was directly or indirectly generated by the attack, we reduce the load on the administrator significantly. We do not claim that DiskDuster is perfect. While it represents a significant improvement over the state of the art, and often restores systems automatically, we require human intervention in some cases. Still, even here DiskDuster indicates in detail which (parts of) files need further scrutiny.

Assumptions In this paper, we assume the following:

1. Intrusions occur at arbitrary points in time and may not be detected until later.
2. Attacks can infect both user processes and the kernel.
3. Attacks may hide themselves root-kit style and turn off AV scanners and other defensive mechanisms on the guest OS.
4. DiskDuster can detect the attack and trace it back to the moment of infection. Given a recorded execution trace, we believe this is a reasonable assumption. A rootkit may hide itself, but it cannot remove itself from the execution trace, which means that AV scanners, taint trackers and other detection methods have a chance to detect it eventually. Once an AV scanner detects a trojan on the system, we skip backwards through the trace until we find a snapshot without the trojan binary, and then replay the execution until it is created and executed for the first time.
5. Attacks cannot tamper with the recording process undetected. As the recorder runs at the level of virtual hardware, this is a reasonable assumption.

Decoupled security While it is *possible* to run DiskDuster on a stand-alone system, we designed it for decoupled security [4]. Decoupled security records the execution on a live production machine and replays it on a dedicated security server with additional security checks and recovery operations (see Fig.1). In other words, all security checks and recovery operations run on the server.

Decoupled security hides the overhead of security checks from the production system. At a small, constant cost of recording on the production system, we can apply

any security check on the replay side, including those too expensive to run on production systems. Since we use full-system DTA, the overhead of our analysis is very high (about 20x), we prefer to run it devolved from the production machine. Also, it is not possible for malware to hide 'rootkit-style' after the infection. As the initial point of infection is preserved in the execution trace, it can be found by periodic rescanning of older traces with new signatures. Finally, recording provides automatic backup, fine-grained versioning, and audit trails. Not surprisingly, decoupled security has become a popular security model [8, 4, 24, 33]. Moreover, vendors like VMWare now offer record and replay functionality in their products [31].

Of course, recording and storing execution traces is not free, but in practice, the costs are low (a few percent increase of CPU overhead and minimal log sizes [4, 24]). A more serious drawback of decoupled security is that attacks are always detected *a posteriori*. The same is true for traditional AV scanners. If a new trojan comes out, it takes a while before AV databases contain a signature for it. In either case, the challenge is to clean up the system and remove all traces of the attack.

Implicit flows One of the most difficult problems for dynamic taint analysis is that of implicit flows [2, 26], and we do not pretend to solve it in this paper. An implicit flow occurs when an assignment depends on a tainted value in a condition. For instance, consider the following code:

```
int y=0; if (x==1) y=1;
```

If `x` is tainted, perhaps `y` should be tainted also? After all, its value is completely determined by `x`. The problem is that implicit flows often lead to overtainting [2, 26]. Recent work by Kang et al. [13] presents an interesting approach to curtail overtainting for certain applications, but for now implicit flows cannot be handled reliably. We do not try to solve them at all, but we cannot afford to ignore them either, as skipping them leads to false negatives. DiskDuster simply takes a conservative approach for malicious data on disk; whenever a process has read malicious or suspicious bytes, all subsequent writes are marked 'suspicious'. As a result, taint laundering is impossible. We discuss more interesting/problematic scenarios related to implicit flows in Section 5.

3 Architecture

After detecting an attack, DiskDuster traces it back to find the point of infection. It then uses DTA to track the malicious code's actions and undo the malicious effects. DiskDuster tries to remove these effects by restoring the disk to a pre-infection state, while presenting a user with lists of files and folders that became malicious or suspicious in the period between the attack and the detection. While the user can safely assume that files classified as benign are intact, they need to scrutinize the suspicious ones. Refer to Fig. 3 for a timeline illustrating the course of actions performed by DiskDuster.

In addition, DiskDuster supports investigators by analyzing the attack. For instance, for drive-by-downloads, DiskDuster separates the shellcode from its packers, and when the shellcode downloads malware, it traces what bytes on disk change.

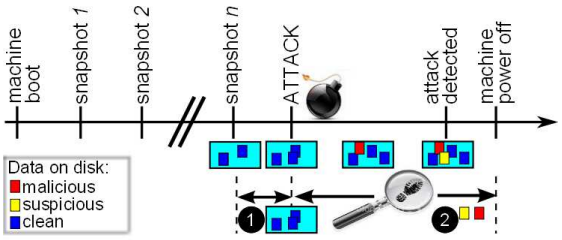


Fig. 3. DiskDuster timeline. Upon detecting an attack, DiskDuster restores the disk to a pre-infection state ①, removes all malicious data, and presents a list of all suspicious files/folders ②.

3.1 Decoupling DiskDuster: recording and replaying execution traces

Recording and replaying executions is hardly novel. In our lab, we have implemented and written about several such systems ourselves, both at full-system [10] and process [24] granularity. Others built similar solutions [8, 1, 30, 19, 33]. Moreover, VMware Workstation 6.5 introduced replaying as standard feature.

By recording only a minimum of non-deterministic events, the overhead of recording is small both in speed (a few percent) and storage (a few hundred Bps) [4]. Moreover, even with expensive detection methods like DTA, the *lag* between the original execution and the replica is minimal. In fact, the replayer typically has no problem keeping up with the recorder, mainly because it does not need to wait (e.g., for reads from the network or file system, or in idle loops). This is known as 'idle boost'.

While the best fit for DiskDuster is clearly our tailor-made Qemu-based full system replayer [10], we believe that with some effort other recorders, including VMWare's could be used also. Indeed, VMWare showed that one can record on VMWare and replay on Qemu in Aftersight [4]. In this paper, however, we focus on recovery.

3.2 Tracking, logging, and snapshotting

Figure 4 illustrates the DiskDuster components. We briefly enumerate each of the modules here, and describe them in more detail in subsequent sections. All these modules operate at the level of the emulated hardware and work with unmodified OSs.

Tainting At the core of our architecture is a dynamic **taint tracking** module, capable of tracking data in memory and on disk. The module is based on Argos [23] and the propagation rules are similar to those of TaintCheck [18] and Minos [6]: (a) taint propagates to the destination (register or memory) whenever tainted data is copied, or used as a source operand in an arithmetic operation, (b) we clean the destination whenever an operation has a constant output (i.e., the output does not depend on the instruction's inputs), and (c) like most systems, we do not propagate taint on dereferences of tainted pointers.

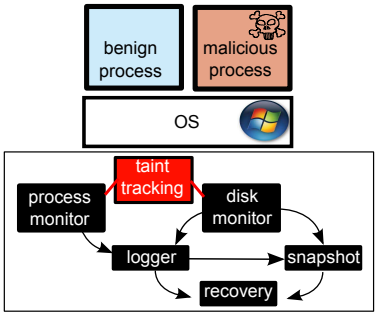


Fig. 4. DiskDuster architecture

Taint tracking in DiskDuster serves two different purposes. First, we use it just like most other DTA solutions—to detect control flow diversion and code injection attacks. For instance, DiskDuster taints all data coming from the network and raises an alarm whenever such bytes modify the control flow of the program directly (e.g., by overwriting the return address). Since we do not track indirect flows, DiskDuster may miss attacks that corrupt memory by means of bytes propagated through indirect transfers, be it tainted dereferences in translation tables, or implicit flows in conditions. It simply means that DiskDuster is not a pefect detector, but we do not see this as a serious limitation. We can easily complement DiskDuster with other detectors, such as AV scanners, but perhaps also more powerful taint trackers. After we detect an attack, the implicit flow is no longer a problem, because we conservatively track everything that could be influenced by malicious data.

Second, and much more essential, is that DTA allows us to monitor malicious programs. For instance, once we know that a process is malicious, we mark all bytes written by the process as malicious also—until we reach the end of the execution trace. Doing so allows us to separate good data from bad data at recovery time. These two uses require different types of taint. Thus, besides the clean/untainted tag, we distinguish *three* types of taint in DiskDuster, corresponding to three sources of taint:

Untrusted ($\bar{U}$) We assign $\bar{U}$ tags to all data from untrusted sources (like the network).
Malicious ($\bar{M}$) We assign $\bar{M}$ tags to all bytes written by malicious processes.
Suspicious ($\bar{S}$) When a benign process reads $\bar{M}$ bytes, we propagate that tag through the execution (see above). Thus, all writes of $\bar{M}$ bytes to disk are also tagged $\bar{M}$. However, even if the written data does not derive directly from $\bar{M}$ data, it may have been influenced by it via an implicit flow. Thus, after a benign process reads $\bar{M}$ or $\bar{S}$ bytes, we label all its writes not already tagged $\bar{M}$ with the $\bar{S}$ tag.

Monitor modules The two monitor modules trace both process execution and disk input/output. Specifically, the **disk monitor** keeps track of all reads and writes to disk, while the **process monitor** tracks running processes. When DiskDuster detects an attack, it marks the compromised process as 'malicious' and notifies the disk monitor. From now on, all writes by this process receive an $\bar{M}$ tag. In the process monitor, malicious processes are handled in a special manner. For instance, when they start a new process, the created process will be marked malicious also.

Logging, snapshotting and recovery The **logger** stores all information generated by the two monitor modules. The logs always include each write and read on disk, and in the case of a compromised process, they also contain detailed information about the context of the process. The **snapshot** module takes snapshots of the disk drive according to user-specified policies. Snapshotting allows us to skip backwards and forwards through an execution trace quickly. The **recovery** module, finally, sanitizes the system by replaying write operations from the last snapshot until the moment of infection.

3.3 Attack detection

In our current prototype, DiskDuster detects attacks in one of two ways. First, it detects memory corruption and code injection attacks by means of dynamic taint analysis. The

process is similar to other full system taint trackers like Argos [23] and Minos [6]. All data arriving from the network is marked 'untrusted' ($\bar{U}$). Whenever such data modifies a process' control flow (e.g., when it ends up in the program counter), the process monitor treats it as an attack.

Second, when an external AV scanner detects new malware, we explicitly contact the process monitor to mark the corresponding process as malicious. The AV scanner is useful for attacks that do not compromise an existing program. For instance, a trojan installed by the user. Other detection methods can be plugged in easily. Regardless of how we detect the attack, from that point onwards, the process monitor tracks the malicious process.

3.4 The process monitor: tracking attacks at thread granularity

Upon detecting an attack, the process monitor closely monitors the offending process(es) to track which files and processes it influences and how. In the process, the process monitor classifies threads and processes as malicious, suspicious or benign. First, we explain these categories, and then we focus on technical challenges to support them. By default, all processes and threads are *benign* and the only exceptions are the malicious and suspicuous threads listed below.

Malicious threads DiskDuster marks all processes corresponding to attacks reported by the AV scanner or DTA module as *malicious*. DiskDuster also treats a thread as malicious if it is attacked by local processes—for instance, when it uses a DLL provided by a malicious process, or when its parent is malicious. Once a thread has become malicious, all its writes are labeled with the $\bar{M}$ tag. We say that a process is malicious if it has a malicious thread.

Thus, the process monitor should both identify the malicious thread, and inspect its execution context, such as the loaded dynamic libraries. Additionally, it tracks the creation of new processes by malicious threads and marks them malicious also.

Suspicious threads As discussed in Section 2, accurate tracking of implicit dependencies is difficult, if not impossible. However, ignoring them causes false negatives. We take a conservative approach, and track *suspicious* threads—threads *possibly* influenced by malware—and ask users to verify the contents of suspicious files during recovery.

A benign thread becomes suspicious when we can no longer guarantee that malware does not influence its actions. First, whenever a benign thread has read a suspicious or malicious byte using I/O routines (e.g., from a file, the registry, or through interprocess communication), we consider it suspicious. Second, when a process has a malicious thread, we cannot rule out implicit flows between the malicious and benign threads. DiskDuster therefore considers all benign threads in this process suspicious. Finally, a child of a suspicious thread is also suspicious. We label all ostensibly clean data written by suspicious threads with the $\bar{S}$ tag. We call any process with suspicious threads suspicious also.

Thus, the process monitor again collects all information necessary to identify a suspicious thread, and tracks the creation of its children. It also monitors the data passed through the I/O routines (e.g., file reads and writes).

Low level tracking to classify processes Since the monitor resides at the (emulated) hardware level, process tracking is not trivial—normal process semantics as defined by the operating system are not readily available. The problem of extracting high-level semantic information from low-level data sources is known as the semantic gap, and has sparked much research activity in recent years [7, 21]. We now discuss how DiskDuster bridges it.

Process and thread identification To identify threads and processes at the level of a processor emulator, we use the solution proposed in Antfarm [12]. It tracks changes of the `cr3` (or page directory base) register, which stores the physical address of the page directory. As a rule, a context switch implies changing the set of active page tables, and thus loading `cr3` with the value stored in the descriptor of the process to be executed. DiskDuster uses `cr3` as a unique process identifier.

However, since all threads of a process share the page table directory, this mechanism does not distinguish between threads. To increase the granularity of tracking, the process monitor additionally looks up kernel-level data structures that hold process information. In 32-bit Windows, the Thread Environment Block (TEB), pointed to by the `FS` register, stores information about the currently running thread. As DiskDuster can easily reach this data structure from the emulator using the register, we extract all relevant thread information directly from the TEB.

Tracking semantics In 32-bit Windows, the process monitor tracks the necessary semantic information by intercepting a number of functions from the `kernel32` library. These include the process creation functions, and the I/O routines, such as the file and registry read functions, or the interprocess communication functions. To determine addresses of these functions, the process monitor implements a solution typically used by shellcode. Using the TEB, DiskDuster identifies first the Process Execution Block (PEB), and then the loaded modules. Each loaded module contains the addresses and symbol names of available functions. DiskDuster uses this information during calls, jumps, and returns, and checks (at the level of the emulated hardware) if the program counter indicates the entry point of a function we intercept. If so, it calls a registered hook.

3.5 The disk monitor

As illustrated in Fig. 3, the disk monitor tracks all reads and writes to disk to support two of DiskDuster's main tasks: (1) restore the disk drive to a pre-infection state, (2) for all post-infection disk activity, present the user with an analysis of clean and suspicious files (so that she can safely keep the clean ones, and verify the suspicious ones).

The first task requires a replay of all disk writes that took place in the period between the last uncorrupted snapshot and the attack. The disk monitor simply logs all operations which modify data on disk, so that they can be repeated later.

Since the analysis phase requires precise information about clean, suspicious, and malicious parts of the disk, DiskDuster extends its taint tracking module to handle disk operations, and stores taint values of the disk contents in a disk shadow map. Whenever a process stores data to disk, the disk monitor checks whether it should label these bytes

with a tag. If the process is listed as suspicious or malicious, the data is labelled with $\bar{S}$ or $\bar{M}$, respectively. Similarly, if the bytes carry a $\bar{U}$, $\bar{S}$, or $\bar{M}$ tag already, DiskDuster simply propagates it to the disk map. Conversely, when the program reads data from disk, the disk monitor propagates tags from the disk map into the main memory map. For instance, when a program reads tainted bytes from disk into memory, DiskDuster tags the corresponding bytes with a tainted tag in the memory map.

The diskmap can store the disk taint information at block level or at byte level, depending on the user's needs. The block level would provide information about which files were touched by an attack, while the byte level would be more specific and show which exact bytes in the files were changed by the malicious process. In the evaluation we used a byte level map. The taint propagation between the disk map and the main memory map is done at the level of the IDE emulator of the VM.

3.6 Snapshots

Once DiskDuster detects an attack, it reverts the disk to a pre-infection state by replaying disk writes that took place before the infection. Since replaying the execution from boot time would incur a high overhead, DiskDuster uses disk snapshots. Upon detecting an attack, it searches for the last snapshot before the infection, and replays only the disk writes that happened since. DiskDuster's snapshots are subject to simple policies, like "snapshot at fixed time intervals", or "snapshot after n disk writes". For our experiments, we use the second option, and snapshot when the total number of writes equals 10% of the disk size[1]. In practice, this occurred approximately every 10 hours.

Suspending the execution may lead to undesired consequences, such as time-outs on network connections. To avoid such problems, DiskDuster implements live snapshots of disk drives. Once it triggers a snapshot, DiskDuster creates a copy of the drive in the background while the VM keeps running. During this process, all writes to disk generate a copy of the modified blocks to the snapshot before they are committed to disk.

3.7 Recovery and analysis

System recovery begins when the user has shut down the machine. As illustrated in Fig. 3, DiskDuster starts by reverting the disk to a pre-infection state, then closely monitors infected processes to find out which files, folders and processes they influenced.

First, DiskDuster determines the initial intrusion moment, or more specifically, the first disk write by a malicious process. In the case of an attack detected by DTA, the intrusion moment is fixed exactly at the point where a good process turns into a bad one. If an AV software detects the infection, DiskDuster scans the logs for an operation that modifies a disk block corresponding to a file matching the AV signature. In either case, DiskDuster replays disk operations which took place before the first malicious write, and it provides the user with a disk in a benign state.

Next, DiskDuster monitors offensive processes to log the data they influence (Sections 3.4-3.5), and presents a user with a list of clean, and suspicious ($\bar{S}$) files (malicious

[1] The blocks need not be unique, so we snapshot also if the same 2GB of a 200GB drive are written 10 times.

data is removed automatically). To map clean and suspicious disk blocks to file names, we use an ntfs library [25] to read the filesystem metadata from the physical disk. We extract the semantics of the filesystem to find the file name corresponding to a block.

As the disk monitor works at the physical level, a block on disk can be located in one of the following regions: (a) inside the filesystem and belonging to the data runs [2] of a file, (b) inside the filesystem but in a free region (i.e. not used by any file), (c) in the filesystem's metadata, or (d) outside the filesystem (e.g., the region of physical sectors 0-63 used by the bootloader). The reason for tracking writes outside the file system regions, is that these sectors are used by advanced malware like TDL4.

The DiskDuster resolver provides a list of filenames for the blocks that belong to filesystem objects like files, folders or metadata, and keeps additional information (like region mappings) for the other blocks. Normal users will be interested mainly in the file names, but security professionals may be interested also in the other information.

4 Evaluation

We now evaluate the effectiveness of DiskDuster in recovering from attacks. We do not focus on performance, other than to say that the slowdown of 20x during analysis on the replay side is no worse than that of other full-system DTA solutions [4, 23, 6]. Moreover, the overhead is sufficiently small to make the replay side keep up (in fact, previous experiments in decoupled security show that even with DTA slowdowns of 100x, the replayer keeps up with the production system, because of the idle time boost [4, 10]).

In the remainder of this section, we use DiskDuster to recover from a variety of attacks.

4.1 Experimental Setup

We ran DiskDuster on a machine with an Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz, with 6MB cache, 4GB of RAM memory, and a SATA disk drive. The operating system running on the host was Linux with kernel version 2.6.32. As the guest we ran Windows XP with RAM memory size of 1GB, and a disk drive of 3GB with an NTFS partition stored in the *raw* format.

4.2 Workloads

To evaluate DiskDuster, we observe how well it recovers from an attack which has happened at a point in the past. We assume that malware is active for a while, and observe how much data modified by the user in the time between the infection and the attack DiskDuster can restore. To test with workloads that are both realistic and repeatable, we recorded several real Windows XP sessions using ReMouse[3], and replayed them once the machine has been infected. The workloads contain the active use of a variety of applications, including the Internet Explorer 6.0 web browser, the FoxIT PDF reader,

[2] File content is made of data runs—lists of disk blocks with the actual content of the file

[3] `www.remouse.com`

the standard Windows Picture and Fax Viewer photo editor, and the Notepad++[4] source code editor. For the experiments in Sections 4.3-4.4, we use five workloads, four short ones (denoted `WS-x`), and one long one (denoted `WL-1`). The short ones are one hour each with different activities with detailed descriptions (see below), while the long one captures three working days of a researcher in our lab.

- `WS-1` - the user visits a number of webpages using IE 6.0 and stores the content of several of them, only to reload them from disk later.
- `WS-2` - using the FoxIT PDF reader, the user loads and reads several PDF documents.
- `WS-3` - the user writes and sends an email, downloads several pictures from the web (IE 6.0) and edits them with the Windows Picture and Fax Viewer photo editor.
- `WS-4` - in this session, the user writes a program using the Notepad++ source code editor and makes a drawing using MS Paint. In both cases, the user stores, reloads, modifies and saves the work in several files.
- `WL-1` - in this session, the user engages in a wide variety of activities corresponding to three full days of work.

4.3 Single step attacks

We first run DiskDuster with a set of straightforward attacks that do one or two things only—to verify that it can recover from malicious actions in isolation. For this purpose, we compromised the system using a drive-by-download from Metasploit (version 3.8.0-dev) and ran the following test attacks at the start of the short workloads: `WS-1`,..., `WS-4` (in each case, we "detect" the compromised process after exactly one hour):

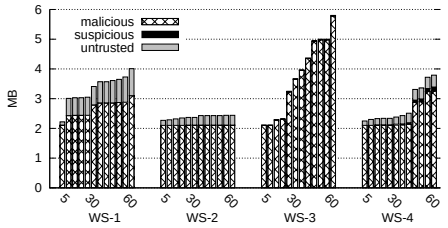
(A) **Binary patch.** The malware binary downloaded modifies the executable file of a benign application (in this case, the IE 6.0 web browser, the FoxIT PDF reader, and MS Paint binaries). DiskDuster performs the analysis, and reverts the binary to its state before the attack.
(B) **Persistent drive-by download.** This time the malware adds a registry key to make itself persistent across reboots. DiskDuster performs the analysis, removes the binary, and restores the registry to its state before the attack.
(C) **File deletion.** The downloaded malware deletes a file from disk. DiskDuster performs the analysis, removes the binary, and reverts the deletion operation.

To evaluate the effectiveness of DiskDuster, we perform two sets of measurements: *infection rates*, and *recovery results*. Infection rates illustrate how quickly taint spreads over the disk. We present the amounts of suspicious, malicious, and untrusted disk data over time. Recovery results show the status of the disk after DiskDuster performed the analysis. We discuss how many benign files and folders a user can safely keep, and how many suspicious ones she needs to scrutinize. We focus on user data, but present results for both `\Documents and Settings`, and `\WINDOWS`.

Fig. 5–7 show the result of these tests. The graphs present the spread of malicious, suspicious and untrusted data on the whole disk over time (at 5 min intervals), while the

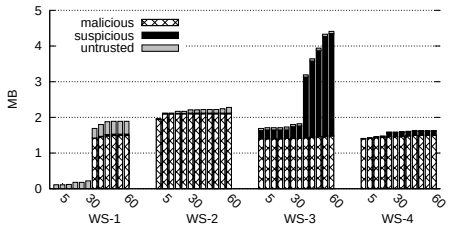
[4] http://notepad-plus-plus.org/

tables count the files containing malicious and suspicious bytes. The files are gathered into two categories: *need review*, and *temporary*. The user needs to scrutinize the former, while temporary files indicate data which she can flush, without any loss of work, for example, the cache and the `History` folder of IE 6.0, or the `dllcache` folder of the `\WINDOWS \system32` directory.



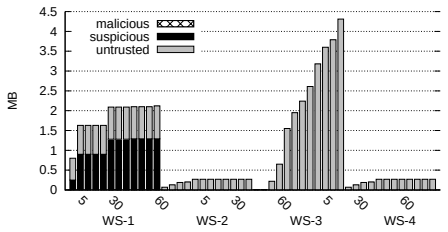
WS	File info	Docum. and Settings				WINDOWS			
		Malicious		Suspicious		Malicious		Suspicious	
		Files	KB	Files	KB	Files	KB	Files	KB
1	Need review	6	6.03	0	0	33	211.72	0	0
1	Temporary	5	19.11	0	0	5	113.37	0	0
2	Need review	0	0	1	4.54	8	169.67	13	1.68
2	Temporary	3	4.37	4	1.64	4	109.85	7	0.61
3	Need review	13	782.23	0	0	38	203.17	0	0
3	Temporary	23	120.07	0	0	5	118.80	0	0
4	Need review	11	374.68	2	250.74	27	193.35	8	35.87
4	Temporary	4	3.82	5	18.81	6	110.21	1	0.46

Fig. 5. The binary patch attack: infection rates and recovery results for four 60 minute workloads.



WS	File info	Docum. and Settings				WINDOWS			
		Malicious		Suspicious		Malicious		Suspicious	
		Files	KB	Files	KB	Files	KB	Files	KB
1	Need review	1	98.85	1	3.83	21	736.85	37	62.61
1	Temporary	8	167.34	116	789.45	1	0.03	2	8.96
2	Need review	0	0	1	10.14	7	168.62	21	6.55
2	Temporary	2	2.51	5	3.00	2	0.18	1	0.18
3	Need review	3	102.27	12	1183.06	25	1440.36	10	56.02
3	Temporary	18	10.34	8	102.67	1	2.17	1	3.43
4	Need review	3	242.12	7	377.20	15	727.69	6	7.50
4	Temporary	7	9.03	3	27.08	0	0	0	0

Fig. 6. The drive-by-download attack: infection rates and recovery results for four 60 minute workloads.



WS	File info	Docum. and Settings				WINDOWS			
		Malicious		Suspicious		Malicious		Suspicious	
		Files	KB	Files	KB	Files	KB	Files	KB
1	Need review	4	1.47	0	0	32	46.33	7	39.00
1	Temporary	71	452.20	1	44.56	1	3.50	0	0
2	Need review	0	0	0	0	0	0	0	0
2	Temporary	0	0	0	0	0	0	0	0
3	Need review	0	0	0	0	1	0	0	0
3	Temporary	0	0	0	0	0.02	0	0	0
4	Need review	0	0	0	0	0	0	0	0
4	Temporary	0	0	0	0	0	0	0	0

Fig. 7. The file deletion attack: infection rates and recovery results for four 60 minute workloads.

We make a few observations. First, `WS-3` is more aggressive in spreading suspicious and untrusted bytes. This makes sense, as the user downloads a fair amount of data and then edits it. All these bytes are at least untrusted, and if the browser was malicious, then all the edits and subsequent writes of these benign processes are suspicious. Second, the three attacks have very different profiles in the way they spread malicious, suspicious and untrusted bytes. This makes sense also, as some attacks make multiple applications malicious, and thus spread more malicious bytes (e.g., the binary patch), while others do not contain much malicious data at all (e.g., the file delete). Finally, we

see that the number of files left malicious or suspicious is small—typically, these are files downloaded by a malicious process and processed by another process. Most of the user data was recovered.

4.4 Complicated, real-world attacks

In this section, we use DiskDuster to recover from four complex attacks involving real world malware, including the Win32/Sality virus [29], the Win32/Alureon trojan [17] and the Win32/Hupigon backdoor [16]. We follow the advanced attack scenario of Fig. 2, and test four malicious binaries in step 4. After launching an attack, we replay the long workload `WL-1`, which captures three working days (Section 4.2). We again detect the attack at the end of the workload, prompting DiskDuster to start its analysis.

In each experiment, the user first infects IE 6.0 by visiting a malicious website. We use Metasploit's meterpreter to migrate the attack from the browser to another application (e.g., the calculator). It deletes the antivirus program, downloads new malware to disk, and executes it. Apart from its normal malicious activities, the malware adds a registry key to make itself persistent across reboots, encrypts the Documents folder on disk, for ransom purposes, and deletes itself from disk[5]. In all cases, DiskDuster was able to restore the disk, undo the encryption, recover the AV scanner, etc.

In the remaining part of this section, we discuss the tested attacks in detail.

(I) **Hupigon backdoor** Win32/Hupigon [16] is a backdoor, which provides an attacker with access to, and control of, an infected machine. Hupigon registers its component as a service.

(II) **Sality virus** Win32/Sality [29] infects executable files. It replaces the original host code at the entry point of the executable to redirect execution to the polymorphic viral code, which has been encrypted and inserted in the last section of the host file. In addition, W32/Sality searches for specific registry subkeys to infect the executable files that run when Windows starts.

(III) **Alureon trojan** Win32/Alureon [17] is a trojan that allows attackers intercept Internet traffic in order to gather confidential information such as user names, passwords, and credit card data. It may also allow to transmit malicious data to the infected computer.

(IV) **Zhelatin email worm / rootkit** Zhelatin [9] spreads in e-mails with war-related subjects as an attachment named "video.exe", "movie.exe", "click me.exe" and so on. After start-up, it drops a randomly named file into the same folder where it was started from and runs it; this file installs a rootkit and p2p (peer-to-peer) component into the Windows System folder. In addition, it kills processes corresponding to virus scanners.

Fig. 8-13 show the results of these tests. The graphs present the spread of tainted data on the whole disk over time. The tables count the files containing malicious and suspicious bytes for two attacks which perform lots of system activities: the Win32/Sality virus and the Win32/Zhelatin email worm/rootkit. Observe that similarly to Section 4.3,

[5] In record/replay, AV scanners can still detect it, as the full execution trace is available.

taint spreads quite aggressively, and it is again expected. For example, all files a malicious IE 6.0 process stores in the `Temporary Internet Files` folder, become malicious as well. Next, since DiskDuster reverts the disk to a pre-infection state (while keeping most recent changes in the user directory), we are not so concerned about the taint in the system files. Finally, observe that the number of other files is small—these are again typically files downloaded by a malicious process, and modified by another one.

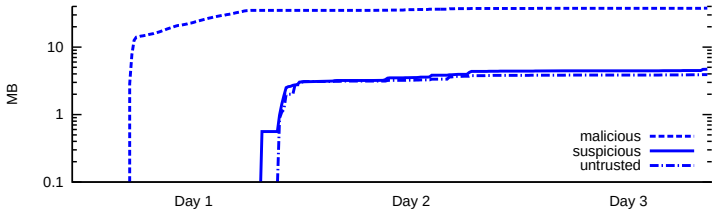


Fig. 8. The Win/32 Hupigon backdoor: infection rates for the `WL-1` workload.

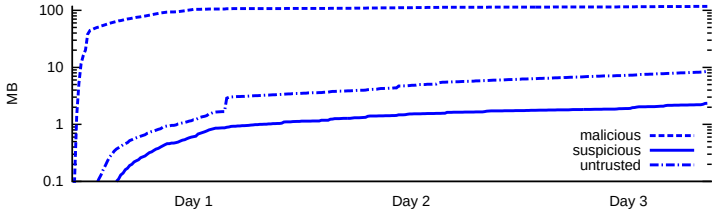


Fig. 9. The Win/32 Sality virus: infection rates for the `WL-1` workload.

Documents and Settings\diskduster					**WINDOWS**				
File info	Malicious		Suspicious		File info	Malicious		Suspicious	
	Files	KB	Files	KB		Files	KB	Files	KB
Need review									
My Documents	11	25253.30	0	0	repair	2	14.33	1	0.21
Local Settings \Application Data	3	7.18	0	0	Microsoft.NET	24	104.66	1	3.21
Documents and Settings	9	950.50	1	53.98	system32	53	927.13	7	16.30
-	-	-	-	-	WINDOWS	32	173.08	6	113.24
Temporary files									
Local Settings \Temporary Internet Files	65	566.79	1	16	system32 \dllcache	14	87.18	1	1.19
Local Settings \Temp	34	74.75	2	6.52	-	-	-	-	-
Local Settings \History	3	23.85	1	0.62	-	-	-	-	-
Recent	5	4.16	1	1.19	-	-	-	-	-

Fig. 10. The Win/32 Sality virus: recovery results for the `WL-1` workload. L S = Local Settings; T I F = Temporary Internet Files

5 Limitations

As DiskDuster automatically recovers in the majority of cases and for very complicated attacks, a valid question is: why not in all cases, and why do we recover user data only—rather than the full system state? The answer is that there are subtle scenarios that are problematic or impossible for DiskDuster to solve. They are related to implicit flows.

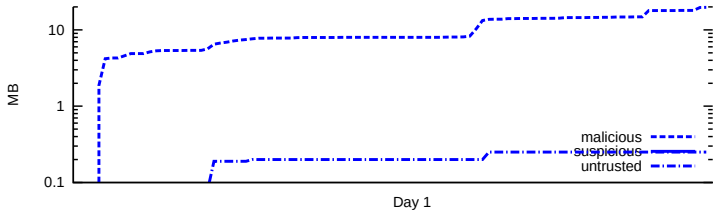


Fig. 11. The Win/32 Alureon trojan: infection rates for the `WL-1` workload. (Due to some problems with the replaying software, we limit the results to one working day.)

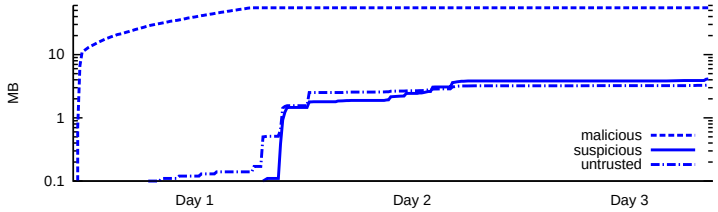


Fig. 12. The Win/32 Zhelatin email worm/rootkit: infection rates for the `WL-1` workload.

The first problematic scenario concerns implicit flow in the user's head. We already discussed it in Section 2: a user makes a note (in a memo, say) about the absence of an AV scanner. As the information flows via the user's mind, DiskDuster cannot detect it.

The second problematic scenario concerns checksums on data structures with malicious data. While not too common, the system occasionally performs a calculation over data structures that contain malicious ($\bar{M}$) data. For instance, consider an OS-level linked list with a checksum. Both malicious processes and benign applications add nodes to the list and update the checksum. Whenever DiskDuster detects malware, the recovery process removes all malicious nodes from the list. However, doing so corrupts both the list and its corresponding checksum. The correct action would be to remove the malicious nodes and all nodes dependent on the malicious nodes, and then to restore the checksum. This is not possible without detailed semantic knowledge about the list. Unfortunately, since the OS sometimes stores such data structures on disk, we may end up with a corrupt system.

To ensure correctness in the presence of implicit flows, DiskDuster currently restores the entire file to an benign version if any part of the file is flagged malicious. This results in correct recovery, but drops more benign writes than strictly necessary.

Finally, there may be implicit dependencies on restored files. Consider again a linked list manipulated by both malicious and benign processes and a benign process that reads a few benign nodes from the list and writes them out to a log. As it does not

Documents and Settings\diskduster					**WINDOWS**				
File info	Malicious		Suspicious		File info	Malicious		Suspicious	
	Files	KB	Files	KB		Files	KB	Files	KB
Need review									
My Documents	1	17.50	1	4.80	repair	0	0	2	8.75
UserInfo	0	0	1	32.06	Microsoft .NET	3	28.00	15	57.85
Cookies	0	0	2	72.95	system32	23	1283.74	13	43.07
NTUSER.DAT	1	1.20	0	0	WINDOWS	16	122.55	20	105.78
Temporary files									
Local Settings\Temporary Internet Files	0	0	50	702.98	system32\dllcache	9	77.00	2	0.69
Local Settings\History	0	0	2	13.99	-	-	-	-	-
Local Settings\Application Data	0	0	1	1.59	-	-	-	-	-
Recent	0	0	5	3.88	-	-	-	-	-

Fig. 13. The Win/32 Zhelatin email worm/rootkit: recovery results for the `WL-1` workload.

read $\bar{M}$ or $\bar{S}$ data, it remains benign throughout its lifetime. At some point, DiskDuster restores the file with the linked list to a previous safe state, as explained above.

The problem is: what do we do with the benign process' log file? Because of the implicit dependency on the file (and its malicious contents), we cannot keep it as is, lest we introduce inconsistencies. Thus, we track the fact that the read accessed a file that DiskDuster restored, thus making the log file a candidate for restoration also. And so on. The additional roll-backs keep the system consistent, but again lead to possibly dropping a few more benign writes than strictly needed.

6 Related Work

Decoupled security checks Recording and replaying is used in many research projects [8, 1, 19]. Full decoupled security for virtual machines was introduced by VMware [4], and the model was quickly picked up by others (e.g., for mobile phones [24] and fast Xen VMs [33]). AfterSight [4] comes closest in spirit to the record and replay side of DiskDuster. However, all these systems differ from DiskDuster in that they limit themselves to attack detection and leave remediation to the administrator.

Data recovery Most automated attack recovery systems either focus solely on data on disks (much like advanced versioning systems), or rely on the support of the target OS—either in the form of a module inside the victim's machine [15, 5, 11, 28, 32, 20, 14, 3], or a proxy [34, 28].

Many of these projects depend on external methods to indicate the root cause of an infection, and to obtain high level semantics (e.g., the way in which the OS uses the password file, the dependencies between OS-level operations, etc.). Such information facilitates the process of intrusion recovery, and aids in building dependency graphs [15, 11, 32, 14], and behavior models [20]. As a result, the analysis becomes more detailed than in systems which operate at the machine level, like DiskDuster.

However, since we cannot assume the integrity of the kernel of a monitored system, it is possible that attacks hinder the analysis, for example by modifying the logs or the dependency graphs. In contrast, DiskDuster carries out a comprehensive analysis without relying on any kernel support whatsoever, and is still able to recover from very sophisticated attacks. We now discuss the most related projects in more detail.

In Wayback [5] versioning is automatic at the write level: each write to the file creates a new version, so that access to any previous version is possible. Wayback needs knowledge about the filesystem and modifies the monitored system. Similarly, BackTracker [15] is implemented inside the OS and tracks OS objects. It extracts dependencies between different components as the attack evolves and can produce dependency graphs.

Taser [11] uses a kernel module to log kernel operations on processes, filesystems and the network for Linux systems. The analysis is decoupled and assumes that the kernel of the monitored host is not compromised. Using the semantic information it constructs detailed dependency graphs to track data flows.

SEE [28] explores one way isolation for Linux processes—processes do not share the disk, and all their commits are written in different locations. It achieves such isolation by interpositioning at the level of system calls and the virtual filesystem layer,

using copy on write implemented as file copy operations. Essentially, it is a filesystem proxy implemented as kernel modules that creates a shadow drive for the process. At the end of execution, it either commits or discards the changes based on user input. Thus, the user must review *all* changes. With DiskDuster users review only suspicious data, while DiskDuster restores the malicious bytes.

Paleari et al. [20] aim to generate remediation procedures to purge infections from a system, but the system can only recover system state and some system files of Windows, and cannot handle deleted files. The system records system calls executed in the emulated environment and infers behavior models based on sequences of the system calls and their parameters.

Retro [14] is a recovery system for Linux that relies on a kernel module to generate action history graphs. The design assumes that the kernel, filesystem, checkpoints or logs are always safe. Another crucial assumption is that the infection is discovered very quickly, otherwise the graphs become too hard to manage. After detecting an infection, the system reexecutes processes and may block if user input is needed and wait for the input in order to continue. In contrast, DiskDuster successfully recovers from attacks that have been active for days.

Back to the Future [3] removes malware and helps users repair systems after an attack. The implementation is Windows specific and requires significant user interaction. The user needs to define *a priori* which is the trusted data and only modifications of this data are logged. Moreover, the user has to decide what to do whenever an untrusted program interferes with a trusted program. The framework is selective about the monitored system calls and may also decide to terminate a process and inform the user.

7 Conclusion

We have described DiskDuster, an attack analysis and recovery system capable of removing all traces from complicated attacks. DiskDuster relies on execution trace recording, snapshotting, and especially taint analysis to track a malcode's actions. Although an attack may be detected long after the infection, DiskDuster is able to roll back to the initial point of infection and restore the disk to that state.We demonstrated the power of our system with complicated and real-world attacks.

DiskDuster greatly helps the analysis of an attack by the classification of bytes located on the physical drive into trusted, malicious and suspicious(which may be the result of implicit flows). Using DiskDuster, the user can recover all post-attack data which was not touched by the attack and is still clean.

Acknowledgments This work is supported by the EU through project ERC-2010- StG 259108-ROSETTA, DG Home iCode and FP7-ICT-257007 SYSSEC.

References

1. Murtaza Basrai and Peter M. Chen. Cooperative Revirt: Adapting message logging for intrusion analysis. Technical Report CSE-TR-504-04, University of Michigan, 2004.

2. Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *DIMVA*, July 2008.
3. Hao Chen, Francis Hsu, Jason Li, Thomas Ristenpart, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. *Proc. of CCS*, 2006.
4. Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, June 2008.
5. Brian Cornell, Peter A. Dinda, and Fabin E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proceedings of USENIX 2004 (Freenix Track)*, 2004.
6. J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture*, 2004.
7. Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *S&P*, 2011.
8. George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
9. F-Secure. Email-Worm:W32/Zhelatin.CQ. `http://www.f-secure.com/v-descs/email-worm_w32_zhelatin_cq.shtml`.
10. Auke Folkerts, Georgios Portokalidis, and Herbert Bos. Multi-tier Intrusion detection by means of replayable virtual machines. Technical Report IR-CS-47, VU University, 2008.
11. Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. *SIGOPS Oper. Syst. Rev.*, 39:163–176, October 2005.
12. Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006.
13. Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS'11, 2011.
14. Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proc. of OSDI'10, Vancouver, Canada*, 2010.
15. Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.
16. Microsoft Malware Protection Center. Backdoor:Win32/Hupigon. `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?name=Backdoor%3AWin32%2FHupigon`.
17. Microsoft Malware Protection Center. Trojan:Win32/Alureon.FE. `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?name=Trojan:Win32/Alureon.FE`.
18. James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.
19. Daniela A. S. Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix, Wu Zhendong, Su Frederic, and T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *ASID 06*, 2006.
20. Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jon Giffin, and Somesh Jha. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX conference on Security*, 2010.
21. Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proc. of SECURWARE'10*, 2010.
22. Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. A foray into conficker's logic and rendezvous points. In *Proc. of LEET'09*, 2009.

23. G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *ACM SIGOPS EuroSys '06*, 2006.
24. Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proc. of ACSAC*, 2010.
25. The Linux-NTFS Project. http://www.linux-ntfs.org.
26. Asia Slowinska and Herbert Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of ACM SIGOPS EUROSYS*, March-April 2009.
27. Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proc. of CCS'09*, pages 635–647, New York, NY, 2009.
28. Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proc. of NDSS*, 2005.
29. Symantec. W32.sality. `http://www.symantec.com/security_response/writeup.jsp?docid=2006-011714-3948-99`.
30. Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi min Wang, and Roussi Roussev. Flight Data Recorder: Monitoring persistent-state interactions to improve systems management. In *7th USENIX OSDI*, 2006.
31. VMWare. Vmware workstation 6.5 beta release notes. `http://www.vmware.com/products/beta/ws/releasenotes_ws65_beta.html`, August 2008.
32. Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, 2007.
33. Shengzhi Zhang, Xiaoqi Jia, Peng Liu, and Jiwu Jing. Cross-layer comprehensive intrusion harm analysis for production workload server systems. In *Proc. of ACSAC'10*, 2010.
34. Ningning Zhu and Tzicker Chiueh. Design, implementation, and evaluation of repairable file service. In *In The International Conference on Dependable Systems and Networks*, 2003.