DIONE: A Flexible Disk Monitoring and Analysis Framework

Jennifer Mankin and David Kaeli

Northeastern University, Boston, Massachusetts, USA
Department of Electrical and Computer Engineering
{jmankin,kaeli}@ece.neu.edu

Abstract. The proliferation of malware in recent years has motivated the need for tools to detect, analyze, and understand intrusions. Though analysis and detection can be difficult, malware fortunately leaves artifacts of its presence on disk. In this paper, we present DIONE, a flexible policy-based disk I/O monitoring and analysis infrastructure that can be used to analyze and understand malware behavior. DIONE interposes between a system-under-analysis and its hard disk, intercepting disk accesses and reconstructing a high-level semantic view of the disk and all operations on it. Since DIONE resides outside the host it is analyzing, it is resilient to attacks and misdirections by malware that attempts to mislead or hide from analyzers. By performing on-the-fly reconstruction of every operation, DIONE maintains a ground truth of the state of the file system which is always up-to-date—even as new files are created, deleted, moved, or altered.

DIONE is the first disk monitoring infrastructure to provide rich, up-to-date, low-level monitoring and analysis for NTFS: the notoriously complex, closed-source file system used by modern Microsoft Windows computing systems. By comparing a snapshot obtained by DIONE's *live-updating* capability to a static disk scan, we demonstrate that DIONE provides 100% accuracy in reconstructing file system operations. Despite this powerful instrumentation capability, DIONE has a minimal effect on the performance of the system. For most tests, DIONE results in a performance overhead of less than 10%—in many cases less than 3%—even when processing complex sequences of file system operations.

Keywords: Malware Analysis, Instrumentation, File System, Digital Forensics.

1 Introduction

As the arms race between malware creators and security researchers intensifies, it becomes increasingly important to develop tools to understand, detect, and prevent intrusions. The amount of malware has not only proliferated in recent years, but it has also become more sophisticated, employing methods to hide from or mislead malware detection mechanisms. As a result, it is critical to obtain information about malware that is as close to the truth as possible, which often

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 127–146, 2012.
© Springer-Verlag Berlin Heidelberg 2012

means working at the lowest level possible. While researchers have had success using memory introspection [7,20,22], disk I/O instrumentation is also a critical tool for the analysis and detection of malware. Disk-level events provide a wealth of information about the state and history of a system. As a result, a flexible disk I/O analysis and instrumentation infrastructure provides key insight to better understand malware behavior.

In this paper, we present DIONE: A **Disk I/O aNalysis Engine**. DIONE is a flexible, policy-based disk monitoring infrastructure which facilitates the collection and analysis of disk I/O. It uses information from a sensor interposed between a System-Under-Analysis (SUA) and its hard disk. Since it monitors I/O outside the reach of the Operating System (OS), it cannot be misdirected or thwarted by rootkits—even those that have achieved superuser-level privilege. DIONE reconstructs high-level file system operations using only intercepted metadata and disk sector addresses; while this reconstruction is performed with a high degree of accuracy, the performance impact of instrumentation is minimal. DIONE only requires basic disk access information that can be obtained by many types of sensors, including both physical hardware sensors and virtualization-based sensors. It can, therefore, be used to analyze and detect malware that utilizes anti-sandbox or virtualization-evasion techniques, which have become increasingly common [3,8,19].

Rootkits, which can gain administrator privilege in order to control a system and hide themselves and other evidence of infection, often leave behind traces of disk activity, even when they eventually cover their tracks [14]. *Persistent rootkits* make changes to files on disk in order to survive reboots; this may include modifications to OS configuration files and system binaries. Even *non-persistent rootkits*, which reside purely in memory, may still present artifacts of infection through disk activity. This activity could include loading dynamic libraries, log-file scrubbing, and file time-stamp tampering [21].

In a simple world, a disk monitor could reside in the OS, where rich, high-level APIs expose semantics such as files and their properties, as well as the high-level operations which create, delete, and modify them. Unfortunately, this is not practical from a security perspective, as it is a well-understood problem that any malware that has escalated its privilege to the administrator level could then thwart or misdirect any data collection and analysis. For this reason, it is more desirable and secure to move the interposer outside the reach of the OS.

Unfortunately, housing a disk instrumentation engine outside the OS prohibits easy access to high-level constructs and operations. The **Semantic Gap** problem occurs when there is no mapping between low-level information (e.g., disk sectors and raw metadata) and high-level information (e.g., files and their properties). Fortunately, this challenge has been addressed in previous work with open-source libraries and drivers [6,26]. However, the **Temporal Gap** problem, in which low-level events across time must be reconstructed to identify high-level file system operations, has not been addressed in detail.

Unlike many low-level disk instrumentation approaches, DIONE analyzes Windows systems running the NTFS file system. Furthermore, it performs *live*

updating, resulting in a view of the file system that is always up-to-date (except for any delay as writes are flushed to disk). DIONE works by pre-populating its data structures with a reconstructed view of the file system of the SUA. Then, as the SUA runs, DIONE intercepts all disk accesses through the use of a sensor. For each sector accessed, DIONE determines which file it belongs to and whether it has intercepted file contents or metadata. Next, DIONE determines whether the file system state changed (e.g., due to a file being created, deleted, etc.). If so, it updates its high-level view of the file system state. Finally, DIONE determines if any policies apply to that file, and if so, performs the appropriate action.

In this paper, we evaluate the accuracy, utility, and performance of DIONE. We integrate DIONE with a popular virtualization infrastructure in order to investigate the disk I/O of a virtual SUA. We also evaluate the performance of full disk instrumentation and the accuracy of DIONE's live updating capability. Finally, we demonstrate the utility of DIONE by instrumenting real-world malware samples and using the results to identify and analyze the malware.

2 Related Work

Much of the previous work in disk analysis focused on Intrusion Detection Systems (IDSs). Kim and Spafford's *Tripwire* monitored Unix systems for unauthorized modifications to the file system [15]. Tripwire performed file-level integrity checks and compared the result to a reference database. While it worked quite well to discover changes to files, it could only detect modifications between scans. Stolfo el al. also developed a host-based file system access anomaly detection system [27]. They utilized a file system sensor which wrapped around a modified file system to extract information about each file access. Both host-based solutions require a trusted OS. Conversely, Pennington et al. implemented a rule-based storage IDS that resided on an NFS server; their IDS monitored disk accesses for changes to specified attributes and file system operations [21].

While host-based IDSs are problematic because a privileged rootkit can override or misdirect malware detectors, IDSs based on Virtual Machine Introspection (VMI) offer both high visibility and isolation from compromised OSs. Payne et al. proposed requirements to guide any virtual machine monitoring infrastructure, and implemented XenAccess to incorporate VMI capabilities [20]. However, the disk-monitoring in their implementation can only be performed on paravirtualized OSes, such as Linux. Azmandian et al. used low-level architectural events and disk and network accesses in their machine learning-based VMI-IDS, though they did not utilize high-level disk semantics [1]. Zhang et al. presented a storage-monitoring infrastructure very similar to ours [29]. However, their monitoring framework was only implemented for FAT32 file systems, which is far less complex than NTFS and is rarely used in modern systems.

Jiang et al. also implemented a VMI-IDS, called *VMwatcher*, which incorporated disk, memory, and kernel-level events [12]. They too could not analyze the ubiquitous NTFS file system, and instead required that Windows VMs use the

Linux ext2/ext3 file system. The VMI-IDS of Joshi et al. detected intrusions before the vulnerability was disclosed [13]. However, their solution to inspecting disk accesses required invoking code in the address space of the guest itself, and subsequently performing a checkpoint and rollback.

Other researchers have acknowledged the role of disk accesses in malware intrusions by providing rootkit *prevention* solutions. With Rootkit Resistant Disks, Butler et al. provided a hardware-based solution to block accesses to sensitive directories, as long as these directories reside on a separate partition [4]. Chubachi et al. also provided a mechanism to block accesses to disk that could operate on a file-level granularity [9]. Unfortunately, they need to create a sector-based "watch-list" before the system boots and do not have a live updating capability to keep the list current as the system runs.

Previous work has also addressed the role of dynamic analysis and instrumentation for malware forensic analysis and classification, and yields information about disk activities. In-host solutions include *DiskMon* [24], part of the Sysinternals tools, and *CWSandbox* [28]; both provide disk access instrumentation capabilities for Windows systems. Similarly, Janus [11], DTrace [5], and Systrace [23] provide in-host instrumentation for Unix-based systems through system call interposition, also providing the ability to instrument disk accesses.

Given that in-host solutions can be misled or thwarted by advanced malware, more recent work has moved the analysis outside the host. Kruegel et al.'s *TTAnalyze* (later renamed *Anubis*) uses an emulation layer to profile malware, including file system activities, of a Windows guest [18]. Similarly, King et al.'s *BackTracker* uses a virtualized environment to gather process and file system-related events that led to a system compromise of a Linux guest [16]. Krishnan et al. created a whole-system analysis, combining memory, disk, and system call introspection [17]. However, their disk monitoring relies on periodic disk scans to connect blocks to files, and does not perform live updating.

3 Dione Overview

Dione is a flexible, policy-based disk I/O monitoring and analyzing infrastructure. Dione maintains a view of the file system under analysis. A disk sensor intercepts all accesses from the System-Under-Analysis (SUA) to its disk, and passes that low-level information to Dione. The toolkit then reconstructs the operation, updates its view of the file system (if necessary), and passes a high-level summary of the disk access to an analysis engine as specified by the user-defined policies. The rest of this section discusses Dione in more detail.

3.1 Threat Model and Assumptions

In our threat model, the SUA is untrusted and can be compromised, even by by malware with administrator-level privileges that can hide its presence from host-level detection mechanisms.

We assume that there is a sensor that interposes between the SUA and its hard disk and provides disk access information. This sensor can be a software sensor

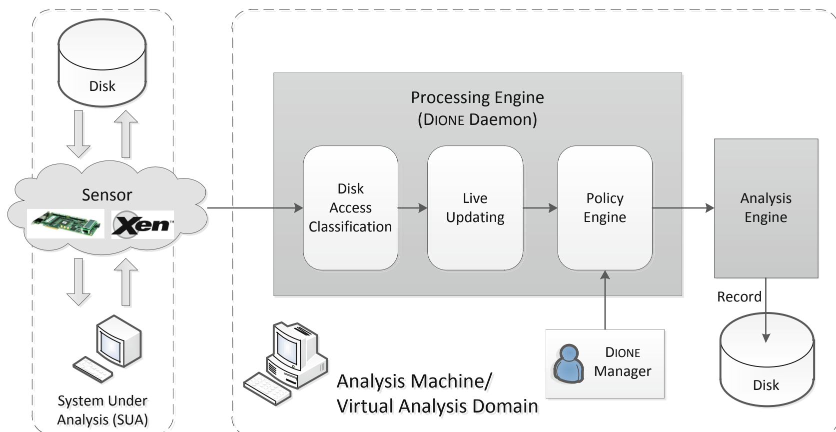


Fig. 1. High-level overview of Dione Architecture

(e.g., a virtualization layer) or a hardware sensor. We assume that both the sensor providing the disk access information and the Analysis Machine (that is, the machine which runs Dione) are trusted. Therefore, in a virtualization-based solution, neither the hypervisor nor the virtual analysis domain is compromised.

3.2 Dione Operation

There are four discrete components to Dione: A sensor, a processing engine, an analysis engine, and the Dione Manager. The Dione architecture is shown in Figure 1.

The **Sensor** interposes between the SUA and its disk. It intercepts each disk access, and summarizes the access in terms of a sector address, a count of consecutive sectors, the operation (read/write), and the actual contents of the disk access (data being read, or data being written). The sensor type is flexible. It can be a physical sensor, which interposes between a physical SUA and the analysis machine, and extracts the disk access information from the protocol (e.g., SATA) command headers to send to Dione. It can also be a virtual sensor, such as a hypervisor, which intercepts the disk I/O of a virtual SUA.

The **Processing Engine** is a daemon on the analysis machine. The multi-threaded Dione daemon interacts with both the user and the sensor. It receives disk access information from the sensor, and performs three steps. The first step is *Disk Access Classification*; for each sector, it determines which file it belongs to (if known) and whether the access was to file content or metadata. In the *Live Updating* phase, it compares the intercepted metadata to its view of the file system to determine if any high-level changes occurred. It passes the high-level access summary to the *Policy Engine*, which determines if any policies apply to the file accessed. If so, it passes the information along to the analysis engine.

Table 1. Commands used for communication with the DIONE daemon

Command	**Description**
DECLARE-RULE	Declare a new rule for instrumentation. Types of rules include: – **Record:** Record an access or file operation – **Timestamp Alert:** Alert if a timestamp is reversed. – **Hide Alert:** Alert if a file is hidden – **MBR Alert:** Alert if the master boot record is accessed.
DELETE-RULE	Delete a previously-declared rule
LIST	List all rules
APPLY	Bulk-apply declared rules to File Record data structures
SCAN	Perform a full scan of a disk image (or mounted disk partition), creating all File Records from the raw bytes and automatically applying all declared rules
SAVE	Save the state of the DIONE File Record hierarchy to a file to be loaded from later
LOAD	Load the DIONE File Record hierarchy from a previously-saved configuration file

The **Analysis Engine** performs some action on the information it has received from the processing engine. Currently, the analysis engine logs the accesses to a file, but future work will extend the analysis engine to perform malware classification or on-the-fly intrusion detection.

The DIONE **Manager** is a command line program which the user invokes to send commands to the DIONE daemon. The commands can be roughly divided into Rule Commands and State Commands and are summarized in Table 1.

Included in the Rule Command category are commands to declare, delete, list, or bulk-apply rules. The policies currently supported are summarized in Table 1. However, DIONE is built to flexibly support the creation of new rule types. The State Command category contains rules to load and save a view of the state of the file system under analysis. The load step is necessary to pre-populate internal DIONE data structures with a summary of the file system. This step is required before DIONE will begin monitoring I/O. The goal of this stage is that DIONE will already know everything about the file system before the SUA boots, so that it can immediately begin monitoring and analyzing disk I/O. This step can be accomplished with a disk scan, which reconstructs the file system from the raw bytes of the disk, or by loading a previously saved configuration file.

3.3 Live Updating

As the SUA boots and runs, new files are created, deleted, moved, expanded, shrunk, and renamed. As a result, the pre-populated view of the SUA's file system, including the mappings between sectors and files, quickly become out-of-date, reducing the accuracy of the monitoring and logging of disk I/O. The solution to this problem is *Live Updating*: an on-the-fly reconstruction of disk events based solely on the intercepted disk access information.

The next sections detail the challenges and solutions to live updating. As our implementation is initially geared toward Windows systems with the NTFS file system, and NTFS is a particularly challenging file system to perform live updating on, we will begin with an introduction to those NTFS concepts which are necessary for accurately describing the live updating implementation.

NTFS Concepts. Many of the challenges of interpreting NTFS arise from its scalability and reliability. Scalability is accomplished through a flexible disk layout and many levels of indirection. Reliability is accomplished through redundancy and by ordering writes in a systematic way to ensure a consistent result.

The primary metadata structure of NTFS is the *Master File Table*, or *MFT*. The MFT is composed of entries, which are each 1KB in size. Each file or directory has at least one MFT entry to describe it. The MFT entry is flexible: The first 42 bytes are the MFT entry header and have a defined purpose and format, but the rest of the bytes store only what is needed for the particular file it describes. In NTFS, everything is a file—even file system administrative metadata. This means that the MFT itself is a file: This file is called *$MFT*, and its contents are the entries of the MFT (therefore, the MFT has an entry in itself for itself). Figure 2 shows a representation of the MFT file, and expands *$MFT*'s entry (which always resides at index 0 in the MFT).

Everything associated with a file is stored in an *attribute*. The attribute types are pre-defined by NTFS to serve specific purposes. For example, the `$STANDARD_INFORMATION` attribute contains access times and permissions, and the `$FILE_NAME` attribute contains the file name and the parent directory's MFT index. Even the contents of a file are stored in an attribute, called the `$DATA` attribute. The contents of a directory are references to its children; these too are stored in attributes.

Each attribute consists of the standard attribute header, a type-specific header, and the contents of the attribute. If the contents of an attribute are small, then the contents will follow the headers and will reside in the MFT entry itself. If the contents are large, then an additional level of indirection is used. In this case, a *runlist* follows the attribute header. A runlist describes all the disk clusters[1] that actually store the contents of the attribute, where a *run* is described by a starting cluster address plus a count of consecutive clusters. In the example MFT of Figure 2, the contents of the `$STANDARD_INFORMATION` and `$FILE_NAME` attributes are resident. Since the content of the `$DATA` attribute is large, this attribute is not resident. Its runlist indicates that the *$MFT* data content can be found in clusters 104-107 and 220-221.

It is easy to see that a small file will occupy only the two sectors of its MFT entry. A large file will occupy the two sectors of its MFT entry, plus the content clusters themselves. Consider, then, the problem of a very large file on a highly fragmented disk: it might take more than the 1024 bytes just to store the content

[1] In NTFS terminology, a cluster is the minimum unit of disk access, and is generally eight sectors long in modern systems.

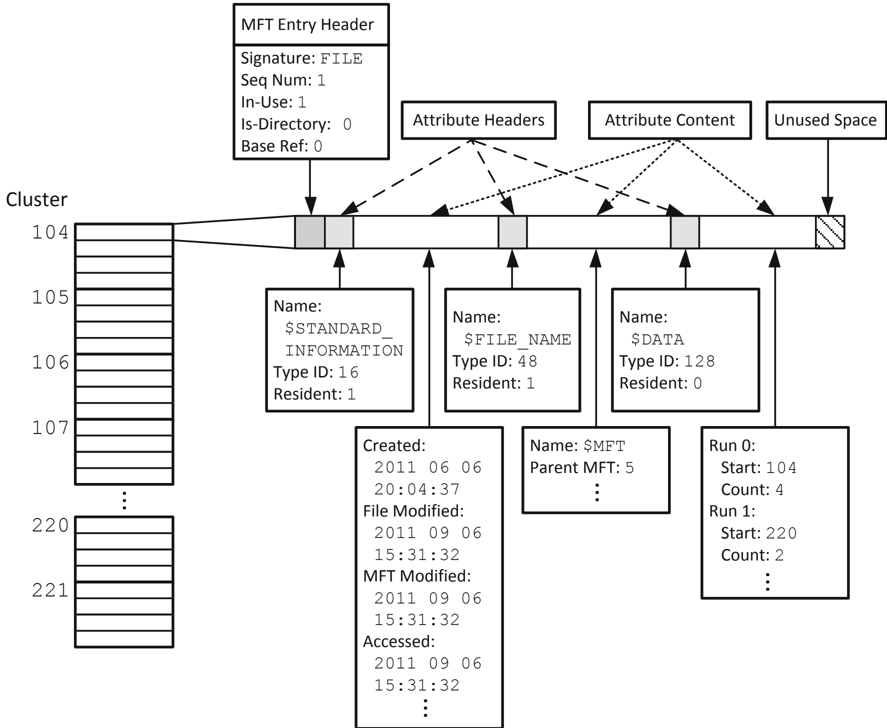


Fig. 2. Representation of the MFT, which is saved in a file called *$MFT*. The first entry holds the information to describe *$MFT* itself; the contents of this entry are expanded to show the structure and relevant information of a typical MFT entry.

runlist. In this case, NTFS scales with another level of indirection and another attribute, and multiple *non-base* MFT entries are allocated (in addition to the *base* entry) to store all attributes.

NTFS Live Updating Challenges. There are two big challenges to live updating: overcoming the Semantic Gap and the Temporal Gap. The Semantic Gap is a well-studied problem in which low-level data must be mapped to high-level data. In our case, we need to map the raw byte contents of a disk access to files and their properties. We utilize and build upon the open-source The Sleuth Kit (TSK) [6] to do much of the work to bridge the semantic gap.

The Temporal Gap occurs when low-level behaviors occurring at different points in time must be pieced together to reconstruct high-level operations. The high-level operations that Dione monitors include file creation, deletion, expansion, move/rename, and updates in MAC times and the hidden property.

The first challenge of live updating is identifying the fields in an intercepted MFT entry for which a change indicates a high-level operation. For some

operations, a combination of changes across multiple intercepted MFT entries indicates that a certain high-level operation has occurred. Due to file system reliability constraints, these changes will be propagated to disk in an inconvenient ordering. Therefore, DIONE must piece together the low-level changes across time in order to reconstruct high-level events.

The biggest challenge resulting from the temporal gap is the detection of file creation. An intercepted MFT entry lacks two critical pieces of information: the MFT index of that entry, and the full path of the file it describes. For a static image, it is not a challenge to calculate both. However, in live analysis, the metadata creation will occur *before* the *$MFT* file's runlist is updated—and just like any other file, *$MFT* can expand to a non-contiguous location on disk. Therefore, it can be impossible to determine (at the time of interception) the MFT index of a newly created file. In fact, it can be impossible to determine at interception time whether a file creation *actually occurred* in the first place.

A similar challenge arises in determining the absolute path of a file. The MFT entry contains only the MFT index of that file's parent, not its entire path. If the parent's file creation has not yet been intercepted, or the intercepted parent did not have an MFT index when its creation was intercepted (due to the previously described problem), DIONE cannot identify the parent to reconstruct the path at the time of interception. This situation occurs quite frequently whenever an application is being installed. In this case, many (up to hundreds or thousands) of files are created in a very short amount of time. Since the OS bunches writes to disk in one delayed burst, many hierarchical directory levels are created in which DIONE cannot determine files' paths.

The temporal gap also proves a challenge when a file's attributes are divided over multiple MFT entries. As DIONE will only intercept one MFT entry at a time, it will never see the full picture at once. Therefore, it needs to account for the possibility of only intercepting a partial view of metadata, and to keep track of non-base entries in addition to base entries.

NTFS Live Updating Operation. Live updating in DIONE occurs in three steps. First, file metadata is intercepted as it is written to disk. Next, the pertinent properties of the file are parsed from the metadata, resulting in a reconstructed description of the file whose metadata was intercepted. Finally, DIONE uses the intercepted sector, the existing view of the file system, and the reconstructed file description from the second step to determine what event occurred. It updates the internal DIONE data structures to represent the file system change.

After intercepting an access to disk, DIONE looks at the intercepted disk contents and approximates whether the disk contents "look like" metadata (i.e., whether the contents appear to be an intercepted MFT entry). If it looks like metadata, DIONE parses the raw bytes and extracts the NTFS attributes. It also attempts to calculate the MFT index by determining where the intercepted sector falls within DIONE's copy of the MFT runlist. With this calculated index, it can attempt to retrieve a File Record. There are two outcomes of this lookup: either a valid File Record is retrieved, or no File Record matches the index.

Table 2. Summary of the artifacts for each file system operation. An MFT *index* is computed based on the intercepted *sector* and the known MFT runlist. If a file record is found with the calculated index, properties of the file record are compared with properties parsed from the intercepted metadata.
* A *replacement* is characterized by a file deletion and creation within the same flush to disk, whereby the same MFT entry is reused.

Operation	Artifacts
File Deletion	– *In-Use* flag **off** in intercepted MFT entry header
File Replacement*	– Creation Time: $Intercepted > FileRecord$, **OR** MFT Entry Sequence Number: $Intercepted > FileRecord$
File Rename	– File Name: $Intercepted \neq FileRecord$
File Move	– Parent's MFT Index: $Intercepted \neq FileRecord$
File Shrink/Expand	– Runlist: $Intercepted \neq FileRecord$, **OR** – Non-base entry created or deleted
Timestamp Reversal	– MAC Times: $Intercepted < FileRecord$
File Hidden	– *Hidden* flag: $Intercepted = 1$ **AND** $FileRecord = 0$

If a valid File Record is found, DIONE will compare the extracted attributes to those attributes found in the existing File Record. If any changes are detected, it will modify the File Record to reflect the changes. A summary of the semantic and temporal artifacts of each type of file operation is presented in Table 2.

If a valid File Record is not found, it means one of three things. In the first case, a new file has just been created, and it has been inserted into a "hole" in the MFT. The file creation can be verified because the intercepted sector falls within the known runlist of the MFT. In the second case, a new file has just been created, but the MFT was full, and thus it could not be inserted into a hole. The MFT index cannot be calculated, because the intercepted sector does not fall in *$MFT*'s runlist. DIONE buffers a reference to this file in a list called the *Wait Buffer*.[2] Eventually DIONE will intercept the *$MFT* file's expansion, the file creation will be validated, and the MFT index and path can be constructed. In the final case, the intercepted data had the format of metadata (e.g., the data looked like an MFT entry), but the data actually turned out to be the contents of another file. This happens for redundant copies of metadata and for the journal file *$Logfile*; additionally, a malicious user could create file contents which mimic the format of a MFT entry. In any of these cases, a reference to this suspected file—and the sector at which it was discovered—will be saved in

[2] A newly-created file will also be placed in the *Wait Buffer* if it has a valid MFT index, but its path cannot be constructed because its parent has yet to be intercepted.

the *Wait Buffer*. However, the *Wait Buffer* will be periodically purged of these File Records when their corresponding sectors are verified as belonging to a file which is not *$MFT*.

The root of trust of DIONE is established and maintained by verifying the location of the MFT during the initial scan or load (the step described in Section 3.2). DIONE maintains a list of all sectors that contain metadata via the runlist of the *$MFT* file. Since that runlist is only updated when *$MFT*'s metadata is intercepted (and the address of this metadata is known and unchanging), the list of sectors containing valid metadata is always verified. Therefore, when data that *looks like* metadata is encountered, it is only processed as metadata if it falls within this list. The only exception to this rule is for new file creation; as discussed above, this case is handled through the Wait Buffer. Therefore, a malicious user cannot forge metadata in order to evade or trick the system.

4 Experimental Results

Next, we evaluate the accuracy and performance of DIONE and demonstrate its utility using real-world malware. Though DIONE is a flexible instrumentation framework capable of collecting and analyzing data from both physical and virtual sensors, we use a Xen-based solution which utilizes the virtualization layer as a data-collecting sensor.

4.1 Experimental Setup

Our virtualization-based solution uses the Xen 4.0.1 hypervisor. Our host system contains a dual-core Intel Xeon 3060 processor with 4 GB RAM and Intel VMX hardware virtualization extensions to enable full-virtualization. The 160 GB, 7200 RPM SATA disk was partitioned with a 25 GB partition for the root directory and a 80 GB partition for the home directory. The virtual machine SUA runs Windows XP Service Pack 3 with the NTFS file system.

Xen uses a QEMU daemon to handle disk requests for a fully-virtualized (e.g., Windows) guest domain; this daemon resides in Domain 0. We implemented a sensor-side API (the *DiskMonitor*), which is linked into the Xen QEMU emulator code. The only modifications necessary to integrate DIONE with Xen are to initialize the DiskMonitor and to call a function when performing a disk access. This function takes as parameters the starting sector address, the consecutive sector count, the operation type, and the actual disk contents that are read or will be written. The TrafficMonitor communicates this information to the DIONE process via shared memory.

4.2 Accuracy Evaluation

In order to gauge the accuracy of live updating, we ran a series of tests to determine if DIONE correctly reconstructed the file system operations for live updating. For our tests, we chose installation and uninstallation programs, as

Table 3. Breakdown of file system operations for each benchmark. The subset of file creations which wait for the delayed expansion of the MFT are also indicated. Note: The "All" test is not a sum of the individual tests, because the OS also creates, deletes, and moves files, and the number of these may differ slightly between tests.

Program	Creations	(Delayed)	Deletions	Moves	Errors
OpenOffice Install	3934	3930	1	0	0
Gimp Install	1380	1380	0	0	0
Firefox Install	152	135	71	0	0
OpenOffice Uninstall	353	62	3788	3836	0
Gimp Uninstall	5	0	1388	0	0
Firefox Uninstall	6	0	80	0	0
All	6500	6114	5986	3815	0

they perform many file system operations very quickly and stress the live updating system. We chose three open source applications (OpenOffice, Gimp, and Firefox), and performed both an installation and a uninstallation for each. We also ran an all-inclusive test that installed all three, then uninstalled all three.

These benchmarks perform a varying number of changes to the file system hierarchy. Table 3 lists each of the seven benchmarks and the number of file creations, deletions, and moves. As discussed in Section 3.3, if many new files are created at once and the MFT does not have enough free space to describe them, there is a delay between when the file creation is intercepted and when the file creation can be verified. We include the number of delayed-verification file creations in Table 3, as these stress Dione's live updating accuracy.

For each test, we started from a clean Windows XP SP3 disk image. We executed one of the seven programs in a VM, instrumenting the file system. We shutdown the VM, and dumped Dione's view of the dynamically-generated state of the file system to a file. We then ran a disk scan on the raw static disk image, and compared the results of the static raw disk scan to the results of the dynamic execution instrumentation. An *error* is defined as any difference between the dynamically-generated state and the static disk scan. This includes a missing file (missed creation), an extraneous file (missed deletion), a misnamed file, a file with the wrong parent ID or path, a file mislabeled as a file or directory, a file mislabeled as hidden, a file with any incorrect timestamp, or a file with an incorrect runlist. Table 3 shows the results of the accuracy tests. In each case, Dione maintained a 100% accurate view of the file system, with no differences between the dynamically-generated view and the static disk scan.

4.3 Performance Evaluation

In order to gauge the performance degradation associated with Dione's disk I/O instrumentation, we ran two classes of benchmarks: one dominated by file content reads and writes, and one dominated by file metadata reads and writes.

Iozone Benchmark. Iozone generates and measures a variety of file operations. It varies both the file size and the record size (e.g., the amount of data read/written in a given transaction). Because it creates very large files, reading and writing to the same file for each test, this is a content-heavy benchmark with very little metadata being processed.

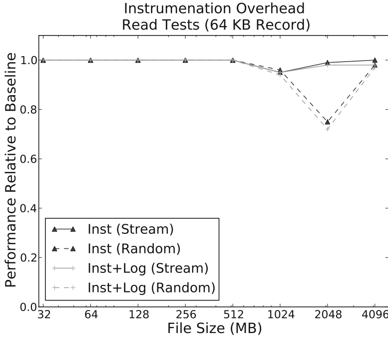
We ran all Iozone tests on a Windows XP virtual machine with a 16 GB virtual disk and 512 MB of virtual RAM. We used the *Write* and *Read* tests (which stream accesses through the file), and *Random Write* and *Random Read* (which perform random accesses). We varied the file size from 32 MB to 4 GB, and chose two record sizes: 64 KB and 16 MB. We ran each test 50 times to average out some of the variability that is inherent with running a user-space program in a virtual machine.

For each test, we ran three different instrumentation configurations. For the *Baseline* configuration, we ran all the tests without instrumentation (that is, with DIONE turned off). In the second configuration, called *Inst*, DIONE is on, and performing full instrumentation of the system. There are, however, no rules in the system, so it does not log any of these accesses. This configuration measures the minimum cost of instrumentation, including live updating. The final configuration is called *Inst+Log*. For these tests, DIONE is on and providing instrumentation; additionally, a rule is set to record every access to every file on the disk. Figure 3 shows the results of the tests. Each of the lines represents the performance with instrumentation, relative to the baseline configuration.

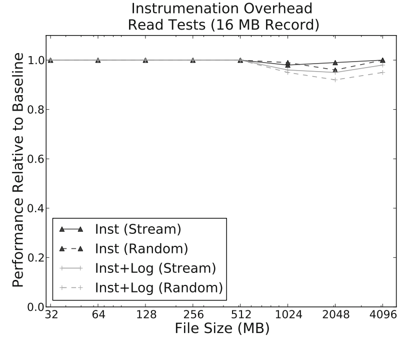
For the *Read* Iozone tests (Figures 3(a) and 3(b)), the slowdown attributed to instrumentation is near 0 for files 512 MB and smaller. Since the virtual machine has 512 MB of RAM, Windows prefetches and keeps data in the page cache for nearly the entire test. Practically, this means that the accesses rarely go to the virtual disk. Since DIONE only instruments actual I/O to the virtual disk—and not file I/O within the guest OS's page cache—DIONE is infrequently invoked.

At larger file sizes, Windows needs to fetch data from the virtual disk, which Xen intercepts and communicates to DIONE. At this point, the performance of instrumentation drops relative to the baseline case. In the worst case for streaming reads, DIONE's no-log instrumentation achieves 97% of the performance of the uninstrumented execution.

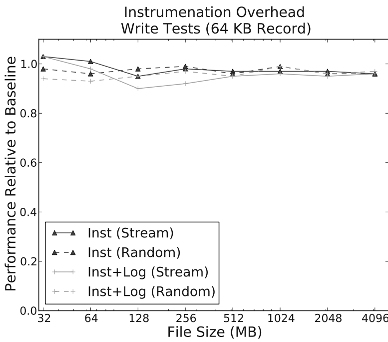
For the random read tests with large file sizes, there is a larger penalty paid during instrumentation. Recall that DIONE incurs a penalty relative to the amount of data accessed on the virtual disk. Therefore, the penalty is higher when more accesses are performed than are necessary. Windows XP utilizes *intelligent read-ahead*, in which the cache manager prefetches data from a file according to some perceived pattern [25]. For random reads, the prefetched data may be evicted from the cache before it is used, resulting in more accesses than necessary. This also explains why the penalty is not as high for the tests using the larger record size (for a given file size). Windows adjusts the amount of data to be prefetched based on the size of the access, so the ratio of prefetched data to file size increases with increasing record sizes. With more prefetched data, there is a higher likelihood that the data will be used before it is evicted from the cache.



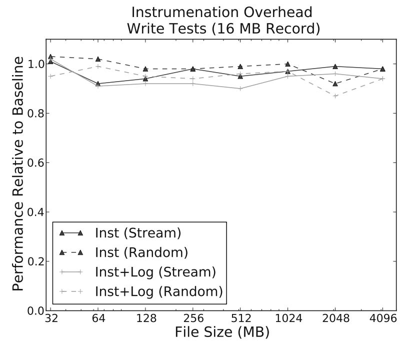
(a) Read Test, 64 KB Record Size



(b) Read Test, 16 MB Record Size



(c) Write Test, 64 KB Record Size



(d) Write Test, 16 MB Record Size

Fig. 3. Performance of instrumentation, normalized to the baseline (no instrumentation) configuration for Iozone benchmarks for streaming and random read and write tests

Fortunately, this overhead is unlikely to be incurred in practice, as random-access of a 2 GB file is rarely performed.

Another observation is that the performance of DIONE actually improves for streaming and random reads as file sizes grow larger than 1 and 2 GB, respectively. This is explained by considering the multiple levels of memory hierarchy in a virtualized system. As the file size grows larger than the VM's RAM, I/O must go to the virtual disk. However, the file may still be small enough to fit in the RAM of the host, as the host will naturally map files (in this case, the VM's disk image) to its own page cache. Thus, disk reads are not performed from the physical disk until the working size of the file becomes larger than available physical RAM. Since physical disk accesses are very slow, any cost associated with DIONE's instrumentation is negligible compared to the cost of going to disk.

The Iozone *Write* tests (Figures 3(c) and 3(d)) show some performance degradation at small files sizes. Windows must periodically flush writes to the virtual

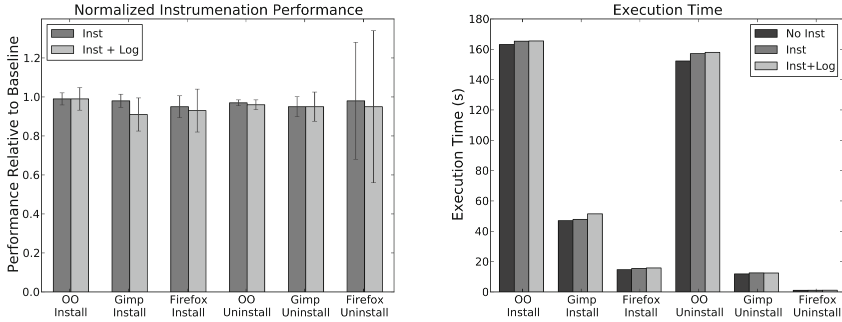
disk, even if the working set fits in the page cache. However, the performance impact is minimal regardless of file size, with a worst-case performance degradation of 10% (though generally closer to 3%). Additionally, the random write tests do not have the same penalty associated with random reads. Since Windows only writes dirty blocks to disk, there are fewer unnecessary accesses to disk.

It is also noticeable that speedup values are sometimes greater than 1 for the 32 MB file size write tests. This would imply that a benchmark will run *faster* with instrumentation than without. In reality, this effect is explained by an optimization Windows uses when writing to disk. Instead of immediately flushing writes to disk, writes are buffered and flushed as a burst to disk. With this *Lazy-Writing*, one eighth of the dirty pages are flushed to disk every second, meaning that a flush could be delayed up to eight seconds [25]. From the perspective of the user—and therefore, the timer—the benchmark is reported to have completed. In reality, the writes are stored in the page cache and have yet to be flushed to disk. Most long-running benchmarks will have flushed the majority of their writes to disk before the process returns. However, a short-running benchmark—such as the Iozone benchmarks operating on a 32 MB file—may still have outstanding writes to flush. The time it will take to flush these will vary randomly through the tests. We reported a 21-24% standard deviation (normalized to the mean) for the baseline, instrumentation, and logging tests. This effect is examined in more detail in the next section.

For all tests, the cost of logging all accesses is relatively low, falling anywhere from 0-8%. For these tests, the root directory (under which the logs were stored) was on a separate partition than the disk image under instrumentation. Therefore, logging introduced an overhead, as the disk alternated between writing to the log file and accessing the VM's disk image. This performance penalty can be reduced by storing the log on the same partition as the disk image. Future work can also reduce the overhead by buffering log messages in memory—performing a burst write to the log—to reduce the physical movements of the disk.

Installation Benchmarks. In the second set of performance experiments, we evaluated the overhead of benchmarks that are high in metadata accesses. These tests will heavily stress the live updating part of DIONE's execution, which comprises the bulk of the computation performed in DIONE. We ran the same six install/uninstall benchmarks as the accuracy tests listed in Table 3. We ran each test ten times to average out variations inherent in running a user-space application on a virtual machine. For each run, we started from the same clean disk image snapshot. We used a Windows XP SP3 virtual machine with an 8 GB virtual disk and 512 MB of virtual RAM.

We compared the baseline execution (with no instrumentation) to full instrumentation with DIONE, with and without logging. Figure 4 shows the execution time for the three configurations, as well as the performance of DIONE's instrumentation relative to the baseline execution. As Figure 4 shows, even when the workload requires frequent metadata analysis for live updating, the overhead of instrumentation is low. Without logging, the full instrumentation of the benchmarks causes a 1-5% performance degradation.



(a) Performance of DIONE instrumentation (error bar equals one standard deviation).

(b) Average execution time with and without DIONE instrumentation.

Fig. 4. Evaluation of DIONE instrumentation for Open Office, Gimp, and Firefox Install/Uninstall benchmarks

The three benchmarks with the least penalty are OpenOffice installation and uninstallation and Gimp installation. These experience between 1-2% performance degradation for instrumentation without logging. Figure 4(b), which graphs the average execution time of the six benchmarks, provides more insight. These three benchmarks are the longest running of the six benchmarks, which is important because of how Windows performs writes to disk. As described in the previous section, writes could be delayed as long as eight seconds before they are flushed from the VM's page cache. While the program is reported to have completed, there are still outstanding writes that need to be flushed to disk. This effect is especially pronounced in any program with a runtime on the same order of magnitude as the write delay.

We can see this effect in Figure 4(a), which includes error bars showing the normalized standard deviation for the 10 runs of each benchmark. The 3 longest-running benchmarks also have the lowest standard deviations. This means that the results of these three tests are the most precise, and the average reflects the true cost of instrumentation. While two of the three shortest-running benchmarks have the highest reported cost of instrumentation, the standard deviation between tests is greater than the reported performance penalty. The execution time of the Firefox Uninstall is dwarfed by the time Windows may delay its writes—as reflected in its high standard deviation. In practice, this means that a user is unlikely to ever notice a slowdown attributed to disk instrumentation for short bursts of disk activity.

The *Inst+Log* tests show a 0-9% performance degradation compared to the baseline. In these tests, the disk image resided on the same partition as the log file. Therefore, the cost of logging to a file was lower than for the content tests.

4.4 Malware Case Studies

Next, we demonstrate the utility of DIONE for forensic analysis by instrumenting two real-world malware samples and using the resulting logs to perform forensic analysis on the intrusion. In each case, we ran unlabeled malware on a clean, non-networked Windows XP virtual machine and instrumented the malware installation and the system upon reboot. We instrumented the entire file system by setting policies to record all accesses, timestamp reversals, and hide-file operations. We analyzed the logs and identified the samples by searching malware description databases based on the resulting file system operations and file names.

We found that DIONE is quite useful for identifying and analyzing malware based on the intentional effects on the file system, such as the creation of files and directories. We also found that it is useful for understanding malware based on the unintentional effects on the file system, such as the loading of system libraries and the creation of system trace files. While the forensic analysis process was performed manually, future work looks to automate this process, using the disk access traces to perform automatic clustering or classification of unknown malware samples.

Backdoor.Bot The first real-world malware sample we discuss is the Backdoor.Bot malware [2], a Trojan first discovered in 2008. This malware opens a backdoor to the infected machine. It creates a directory and a process named *spoolsv*; since spoolsv is also the name of a legitimate Windows process, this malicious process is able to hide in plain sight from the average user. The malware is distributed with an image named *xmas.jpg*.

When the malware is first executed, DIONE observes the creation of several files and directories. First, it creates the top-level directory, *WINDOWS\Temp \spoolsv*. This directory is created with the hidden flag already set, so that it cannot be viewed by the user. In the spoolsv directory, 12 more files are created with their hidden property already set, including the executable *spoolsv.exe*. Six other files are also created without their hidden property set, but that reside in the hidden spoolsv directory anyway. One of these files is the image file *xmas.jpg*; it is displayed to the user after the malware installs to deceive the user into believing that he simply opened an image file.

In addition to detecting infection through the intentional creation of the *spoolsv* directory and its contents, DIONE can also deduce that some meaningful applications are run by the malware though unintentional file system artifacts. In order to speed up the time to load an application, Windows creates a trace file to enable fast future loading of the application. These trace files are stored in the *WINDOWS\Prefetch* directory. Therefore, the creation or access of one of these prefetch files indicates that the corresponding application has been run. DIONE intercepts and records the creation of two prefetch files corresponding to *cmd.exe* and *regedit.exe*. This indicates that the malware has used `cmd` to launch `regedit` to modify the Windows Registry.

FakeAlert Defender. The FakeAlert System Defender trojan, identified by McAfee labs in 2011 [10], is "scareware" that modifies the file system in order to scare the user into purchasing an application to clean his system.

A few seconds after the malware has been executed, the user will see several error messages pop up alerting the user about different types of disk failures. As the user looks through his folders, all files will appear to have been deleted, though all directories remain. When the user reboots, the desktop is black, and it appears as if all files, directories, and even executables are lost. Instrumentation with Dione provides insight into how all of these actions are accomplished.

As the malware is executed, Dione observes that it first renames the original malicious file with a randomly-generated name with the extension *.exe.tmp*. It moves this file to the *Documents and Settings\%user_name%\Local Settings* directory, which is hidden by default. Next, it creates a randomly-named executable in the *Documents and Settings\All Users\Application Data* directory, which is also hidden by default. As it does with any newly-loaded application, Windows creates a prefetch file for the executable in *WINDOWS\Prefetch*.

Next, Dione observes that the malware performs the following steps with the goal of creating a copy of the file system hierarchy in a temporary folder. First, it creates a randomly-named directory in *Documents and Settings\%user name%\Local Settings\Temp*, and some numerically-named subfolders (e.g., *1*, *4*). Within these subfolders, the malware creates new directories, maintaining the hierarchy of the original filesystem. It then iterates through the user's existing file system hierarchy, and moves all files (not directories) into the corresponding directory under the *Temp* folder. The result is a hidden replication of the original hierarchy. While the original directory hierarchy also remains, all folders are empty, so it appears to the user that all his files have disappeared.

Once the user reboots, Dione observes the malware reversing the timestamp on the original malware executable. Finally, the malware iterates through every file and directory in the file system and changes its property to *hidden*, completing the deception that every file and directory on the disk has been deleted.

5 Conclusions

In this paper, we introduced Dione: a flexible, disk I/O instrumentation infrastructure for analyzing the ubiquitous Windows NTFS file system. Disk I/O is intercepted by a sensor, which passes disk access information to Dione for analysis. By residing outside the host, Dione is protected from the malware it is instrumenting. However, Dione has to bridge both the semantic and temporal gaps—not just reconstructing high-level semantics from low-level metadata, but also reconstructing high-level file operations from low-level events. We discussed the challenges of reconstructing disk operations, a process we call *Live Updating*, which ensures that Dione always has an up-to-date view of the file system.

We demonstrated that Dione achieves 100% accuracy in tracking disk operations and reconstructing high-level operations. We showed that despite this powerful instrumentation capability, Dione does not suffer from large performance degradation. We evaluated Dione's performance with workloads that

generate a large volume of content accesses, as well as workloads that generate a high rate of metadata accesses and stress the live updating system. Dione preserves over 90% of the performance of native execution for most tests. We demonstrated the utility of Dione for forensic analysis by instrumenting two real-world malware intrusions. We showed that Dione can detect suspicious file operations that are hidden from the user, including file creations, timestamp reversals, file hiding, and the launching of applications to alter OS state.

Acknowledgments. The authors would like to thank Charles V. Wright and Joshua Hodosh for their invaluable input, guidance, and motivation for the development of Dione. Additionally, we would like to thank Dana Schaa for his continued support and feedback during the development of this work. Finally, we thank the anonymous reviewers for their comments and suggestions. This work is supported in part by a grant from MIT Lincoln Laboratory, and by Northeastern University's Institute for Information Assurance.

References

1. Azmandian, F., Moffie, M., Alshawabkeh, M., Dy, J., Aslam, J., Kaeli, D.: Virtual machine monitor-based lightweight intrusion detection. SIGOPS Operating Systems Review 45 (July 2011)
2. Virus profile: Generic backdoor!68a521cd1d46., http://home.glb.mcafee.com/virusinfo/.aspx?key=199638 (accessed on December 11, 2011)
3. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient detection of split personalities in malware. In: Network and Distributed System Security Symposium, NDSS (2010)
4. Butler, K.R., McLaughlin, S., McDaniel, P.D.: Rootkit-resistant disks. In: Computer and Communications Security (CCS), pp. 403–416. ACM (2008)
5. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: USENIX Annual Technical Conference, ATEC 2004. USENIX Association (2004)
6. Carrier, B. The Sleuth Kit (TSK), http://www.sleuthkit.org (accessed on October 1, 2011)
7. Case, A., Marziale, L., Richard III, G.G.: Dynamic recreation of kernel data structures for live forensics. Digital Investigation 7(suppl. 1) (2010); The Proceedings of the Tenth Annual DFRWS Conference
8. Chen, X., Andersen, J., Mao, Z., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Dependable Systems and Networks (DSN), pp. 177–186 (2008)
9. Chubachi, Y., Shinagawa, T., Kato, K.: Hypervisor-based prevention of persistent rootkits. In: Symposium on Applied Computing (SAC). ACM (2010)
10. Mcafee labs thread advisory: Fakealert system defender. White-Paper, McAfee Inc. (June 2011)
11. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications: Confining the wily hacker. In: USENIX Security Symposium. USENIX Association (1996)

12. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In: Computer and Communications Security (CCS), pp. 128–138. ACM (2007)
13. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: ACM Symposium on Operating Systems Principles (SOSP 2005), pp. 91–104 (2005)
14. Kapoor, A., Mathur, R.: Predicting the future of stealth attacks. In: Proceedings of the Virus Bulletin Conference (October 2011)
15. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: a file system integrity checker. In: Computer and Communications Security (CCS), pp. 18–29. ACM (1994)
16. King, S.T., Chen, P.M.: Backtracking intrusions. In: Symposium on Operating Systems Principles (SOSP). ACM (2003)
17. Krishnan, S., Snow, K.Z., Monrose, F.: Trail of bytes: efficient support for forensic analysis. In: Computer and Communications Security. ACM (2010)
18. Kruegel, C., Kirda, E., Bayer, U.: TTAnalyze: A tool for analyzing malware. In: European Institute for Computer Antivirus Research, EICAR (2006)
19. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting Environment-Sensitive Malware. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 338–357. Springer, Heidelberg (2011)
20. Payne, B.D., de, A., Carbone, M.D.P., Lee, W.: Secure and flexible monitoring of virtual machines. In: Annual Computer Security Applications Conference, ACSAC (2007)
21. Pennington, A.G., Strunk, J.D., Griffin, J.L., Soules, C.A.N., Goodson, G.R., Ganger, G.R.: Storage-based intrusion detection: Watching storage activity for suspicious behavior. In: USENIX Security Symposium (2003)
22. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: USENIX Security Symposium. USENIX Assoc. (2004)
23. Provos, N.: Improving host security with system call policies. In: USENIX Security Symposium, Berkeley, CA, USA. USENIX Association (2003)
24. Russinovich, M.: DiskMon for Windows v2.01, `http://technet.microsoft.com/en-us/sysinternals/bb896646` (accessed on November 24, 2011)
25. Russinovich, M.E., Solomon, D.A.: Microsoft Windows Internals, 4th edn. Microsoft Press (2005)
26. Russon, R., Fledel, Y.: NTFS documentation. Tech. rep., Linux NTFS (2004)
27. Stolfo, S.J., Hershkop, S., Bui, L.H., Ferster, R., Wang, K.: Anomaly Detection in Computer Security and an Application to File System Accesses. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 14–28. Springer, Heidelberg (2005)
28. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security Privacy 5(2) (March-April 2007)
29. Zhang, Y., Gu, Y., Wang, H., Wang, D.: Virtual-machine-based intrusion detection on file-aware block level storage. In: Symposium on Computer Architecture and High Performance Computing. IEEE Computer Society (2006)