Monitoring Access to Shared Memory-Mapped Files

By

Christian Sarmoria, Steve Chapin

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2005 USA

New Orleans, LA (Aug 17th - 19th)

Monitoring Access to Shared Memory-Mapped Files

Christian G. Sarmoria, Steve J. Chapin
Syracuse University
Syracuse, NY 13244, USA
{cgsarmor,chapin}@ecs.syr.edu

Abstract

The *post-mortem* state of a compromised system may not contain enough evidence regarding what transpired during an attack to explain the attacker's *modus operandi*. Current systems that reconstruct sequences of events gather potential evidence at runtime by monitoring events and objects at the system call level. The reconstruction process starts with a detection point, such as a file with suspicious contents, and establishes a dependency chain with all the processes and files that could be related to the compromise, building a path back to the origin of the attack. However, system call support is lost after a file is memory-mapped because all *read* and *write* operations on the file in memory thereafter are through memory pointers. We present a runtime monitor to log *read* and *write* operations in memory-mapped files. The basic concept of our approach is to insert a page fault monitor in the kernel's memory management subsystem. This monitor guarantees the correct ordering of the logs that represent memory access events when two or more processes operate on a file in memory. Our monitor increases accuracy to current reconstruction systems by reducing search time, search space, and false dependencies.

Keywords: event reconstruction, memory mapping, causality, memory access monitor

1 Introduction

After an intrusion detection system alerts the system administrator of a compromise, the forensic investigator collects all available evidence to understand where, how, and when the attack originated. Evidence availability has a direct impact on the certainty of the conclusions drawn by a forensic investigator regarding the attacker's *modus operandi*. An attacker may delete or obfuscate evidence to hide her malicious actions. Moreover, certain events in the system, such as ongoing file modifications, terminated processes, inter-process communications, and memory usage, may not leave *post-mortem* evidence in the compromised system unless a mechanism to record such events is implemented in a runtime monitor. In Unix-like environments, file system runtime activity is a rich source of potential evidence because programs, data, and even hardware resources are abstracted in the file system and accessed through OS system calls.

The process of evidence gathering focuses on recognizing and identifying evidence based on system objects' characteristics [6]. Event reconstruction examines evidence to find out why an object has certain characteristics. Its outcome is the identification of the events that caused the current state of the object as we need to know why an object has become evidence. Previous states of objects need to be known to reconstruct the attacker's steps one by one and draw a theory explaining the incident. Instead of concentrating on the final state of an object, reconstruction systems examine the events that affected the object.

Traditionally, logging systems operate at the application level. They are semantically rich but they log only events and objects involved in the application's execution. Because they run in user space they can also be easily disabled. Network-level logs can bypass the application-level weakness but they are rendered useless by encryption or obfuscation. Using virtual machine technology [13], machine-instruction-level logging systems have been implemented [8]. Although they are not affected by the constraints present in application and network-level logging systems, they are semantically poor and require the time consuming task of re-execution of the instructions for analysis.

OS-level logging provides a solution to most of these weaknesses. A logging mechanism implemented in the OS kernel cannot be easily disabled. Application independence is acquired by logging at the system call level. Backtracker [21] and Forensix [12] are examples of systems that use these mechanism based on logging events and system objects affected. Events such as file *read/write*, process creation, process termination, and so on are captured through system call tracing and logged as potential evidence. Processes, files, and filenames are identified as the objects on which the events produce their effect. A dependency relationship is composed of three parts: a sink object (process reading a file), a source object (file being read), and a time interval where the event took place. The time of each event is very important to establish dependencies between objects. For example, a process who wrote file X at time *10* has no dependency on a process that read that file at time *5*. At the time a system is found to be compromised, a forensic investigator can analyze logged potential evidence to reconstruct the sequence of steps used by the attacker to accomplish her goal. Logging at the OS system call level has the added advantage of being a low level, application-independent event capture point which cannot be affected by encryption or obfuscation. It is also semantically rich because a system call provides detailed information regarding the event. System call parameters can also be used to augment the semantics of the captured events [12].

However, current reconstruction systems do not consider *read* and *write* events to files when these are mapped in memory. A subverted process changing the contents of a shared memory-mapped file may affect the behavior of a legitimate process that reads the modified shared memory area later on. Memory operations cannot be logged by tracing system calls because a process makes use of pointers to reference its memory address space. Current reconstruction systems can only establish an unconditional dependency between two processes that share memory, no matter what type of operations are carried out on it, if any, leading to false positives.

In this paper we present a runtime monitor to log *read* and *write* operations in memory-mapped files. The basic concept of our approach is the use of page faults to monitor memory access. Our technique is implemented in the kernel's memory management subsystem and guarantees the correct ordering of the logs that represent memory access events when two or more processes operate on a file in memory. Because it monitors memory operations at runtime, it does not require access to the source code of the monitored processes. Our monitor increases the accuracy of current reconstruction systems by reducing search time, search space, and false dependencies.

The structure of this document is organized as follows: Section 2 introduces background and definitions for event reconstruction. Section 3 presents our approach to study memory-mapped objects. Section 4 describes our page-level monitor prototype for shared memory mapped files in the Linux kernel running in the x86 architecture. Section 5 summarizes related work. Section 6 describes our current direction and future work. Section 7 presents conclusions.

2 Background

2.1 Digital Forensics

Digital forensics is “the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations” [22]. The success of an investigation process relies heavily on evidence availability, digital forensic tools, and analysis skills of the investigator.

Tools used in digital forensic investigations implement a layer of abstraction through which input data is analyzed and presented at yet a higher level of abstraction [5]. Data is transformed to become more manageable for the investigator. That is a key concept as it provides a meaningful view of the evidence in a compromised system. For example, consider a large network log file: It needs to be abstracted to a higher level in order to devise meaningful activity such as a file download. Many abstraction layers may need to be applied to convert raw data into tangible evidence.

The digital forensics process is composed of the following phases [23]:

Identification: Incident detection. A number of Intrusion Detection Systems (IDS) have been implemented to handle the identification phase, although there are incidents that still need the intervention of a system administrator in order to be detected.
Preparation: Tools and techniques preparation. Approval to conduct investigation.
Approach strategy: Preparation of a strategy to maximize the collection of evidence while minimizing the impact on the victim.

Preservation: Isolation to secure physical and digital evidence.
Evidence collection: Recording of the physical crime scene and duplication of digital evidence. The compromised system may not contain enough evidence available for an investigator to draw conclusions and support theories as a result of the analysis. A running system goes through a large number of different states until it is compromised. What could be valuable evidence may be lost given the volatile nature of certain events and objects in the system such as temporary files, terminated processes, memory, and so on.
Examination: Search for evidence related to the incident.
Analysis: Drawing of conclusions based on significance of evidence collected. Reexecute the examination phase until a theory can be supported.
Presentation: Explanation of the final theory based on conclusions.
Return evidence: Return of all the evidence removed from the crime scene back to the owner.

Permanent storage, such as a hard drive, is a rich source of evidence in a compromised system because changes made by an attacker are likely to be reflected in the file system. Malicious programs, data files, modified configuration files, and replaced system tools can be recovered by the forensic investigator even when the attacker may have done an effort to clean up after herself in order to hide her activity [7]. The file recovery process has no guarantee of success because it depends on how many times the physical area where the deleted file was originally stored was overwritten later on. Therefore, not all the actions carried out by an attacker may leave evidence that can be gathered at the time a compromise is detected. Furthermore, actions such as ongoing file modifications, memory usage, and short-lived processes may not always leave any trace in permanent storage.

2.2 Events and System Objects Definitions

The following definitions introduce concepts to understand the process of real-time gathering of potential evidence [6]:

Digital evidence of an incident: Any digital data that contains reliable information that supports or refutes a hypothesis about the incident.
Object: An entity with its own characteristics and features. Examples of objects are files and processes
State of an object: The value of its characteristics, or data it contains, with the characteristics being what helps to identify the object.
Event: An occurrence that changes the state of one or more objects.
Crime or incident: An event that violates a law or policy.
Event reconstruction: The process of determining the events that occurred at a crime scene by using evidence characteristics.

An object is *evidence* of an incident if its state was used to cause an event related to the incident or if its state was the effect of an event related to the incident.
Objects can be *cause* or *effect* depending on the role they play in an event. A *cause* object has an influence on the effect of an event. It has characteristics that were used in the event, and the same effect would not have occurred if the object were not present. An *effect* object is one whose state was changed due to the cause objects in the event. It is clear that an effect object in one event can be a cause object later on in another event [6]. A file is an *effect* object when modified by a process, and becomes a *cause* object when it is read by a process later on. An *event* can be redefined as an occurrence that changes the characteristics of one or more effect objects by using the characteristics of one or more cause objects. In reconstruction, the goal is to find the initiator of a chain of events that represents the sequence of steps taken by the attacker to compromise the system.

Dependencies are categorized according to the objects involved [21]:

Process-Process dependency: One process creates another process, or shares memory with it, or signals it. For example, a user logs in through ssh, forks a shell process, and launches a denial of service.
Process-File dependency: A file can affect a process through its data or attributes in events such as a read. A process can affect a file by writing onto it.
Process-Filename dependency: Caused by system calls that contain a file name as an argument: `link, chmod, unlink, open, rmdir, rename,` etc.

A ***detection point*** is the state of the system where an intrusion has occurred and the system administrator is notified. It can be signaled by the system administrator herself who observes suspicious activity such as unusual processes, files, connections, and so on. It can also be signaled by automated tools known as Intrusion Detection Systems (IDS): Tripwire [20] detects modified system files; Snort [24] keeps track of network activity; firewalls detect unusual port connections; monitors that detect unusual system calls sequences made by a process [16]. The reconstruction process begins after a detection point has been signaled.

2.3 Memory-mapped files

Modern operating systems provide services to map files in memory. Memory mapped files provide a mechanism for a process to access files by directly incorporating file data into the process address space. Its use can significantly reduce I/O data movement because the file data does not have to be copied into process data buffers as opposed to *read* and *write* operating system services. It also provides a low-overhead mechanism by which processes can communicate. In Linux, the `mmap` system call associates a memory region with some portion of either a regular file present in a disk-based file system or a block device file. As a result of this association, the kernel translates any access to a byte within the page of the memory region into an operation on the corresponding byte of the file. Each process has its own *process descriptor* and all information related to the address space of a process is included in a *memory descriptor* [4, 14]. This memory descriptor contains the *mmap* field which points to the head of a linked list of memory regions. A *memory region* or virtual memory area (VMA) is a set of contiguous memory frames allocated to a process. A process can own one or many memory regions according to its needs. The kernel is responsible for defining the size of a memory region depending on the amount of memory required by the process as well as contiguous memory frames availability. It also provides a protection mechanism with access permissions for the frames of every memory region.

There are two kinds of memory mapping: *shared* and *private*. In a *shared* memory mapping a *write* operation on any page of the memory region will change the file on disk and will also make changes visible to all other processes that map the same file. In a *private* memory mapping, changes made to any page of the memory region will not be reflected in the file on disk nor be visible to other processes that map the same file. Moreover, a *write* operation on a private mapped page will cause it to stop mapping the page in the file. Private memory mapping is meant to be used when a process creates the mapping just to read the file, not to write onto it [4].

When a process makes an `mmap` system call, it must specify if the mapping will be *shared* or *private*, as well as the access permissions (*read* or *write*). Once the mapping is created, the process can access the data stored in the file by simply reading from the memory locations in the corresponding memory region. The same way data can be written to the file when the memory mapping is shared. The previously mentioned *mmap* parameters are kept in flag fields present in the memory region descriptor. When the kernel allocates a new page, it sets the flags of the corresponding *Page Table Entry* (PTE) according to the value of these fields so that checks can be performed by the Paging Unit circuitry to detect page fault exceptions [14].

3 Granularity

Our motivation to fine-grain the monitoring of memory-mapped objects originated in a weakness we found in the approach taken by systems such as Backtraker [21] and Forensix [12] regarding memory mapping. They assume a dependency between two processes exists whenever they share a memory space, no matter whether they read or write onto it. Both reconstruction systems aggregate shared memory accesses into a single event (the `mmap` system call) over a long period of time, and *load* and *store* memory operations are not monitored at all. The result can be a number of false dependencies because it cannot be established if the file object was *cause* or *effect* at any time while it was mapped in memory. A process making a `mmap` system call does not mean that it will read or write to the file. Furthermore, when a process is created it inherits dependencies with each of the files mapped by its parent, no matter if the child process will actually read or write to the memory area.

We consider memory-mapped files as objects with constituent parts such as pages. Our goal is not only to trace *read* and *write* operations to memory, but also to determine what location of the file is accessed. The following table shows four levels of granularity at which we study memory-mapped objects:

Object-level: The whole object such as the memory-mapped file, without considering any subset of it.

VMA-level: The memory regions allocated to the mapped file. Can be one or many, depends both on contiguous memory frames availability and the amount of memory requested by the process.

Page-level: Most operating systems define a page as a memory allocation unit to take advantage of the paging

Constituent Parts	**Size**
Object	Whole size of the object.
Memory Region (VMA)	A set of contiguous memory pages. Size defined at runtime by the OS.
Page	As defined by the OS, usually 4KB.
Minipage	1 byte $\leq$ *minipage* $<$ *page*. Fixed size or defined at runtime.

Table 1: **Levels of Granularity for Memory Monitoring.**

hardware support present in the underlying architecture.

Minipage-level: The concept of *minipage* defined in the Millipage system [19] introduces the idea of hybrid memory units. Although hardware support is lost for memory allocation units smaller than a page, we have already studied techniques to monitor at this level, which we introduce in the future work section of this paper.

4 Monitoring Techniques

4.1 Object-level

Monitoring at this level is the coarse-granularity approach taken by previous works such as Backtracker [21] and Forensix [12]. However they only log the `mmap` system call and do not monitor any further accesses to the shared memory area where the file object resides. Our object-level monitor can determine whether, once in memory, the object was read or modified by any process at any time. The technique we use is the protection of memory pages to generate page faults at the time a process operates on the memory space allocated to the memory-mapped file. Our monitor interprets a page fault as a process' intention to either read or write to the memory-mapped file. The object-level monitor does not consider constituent parts at all. It has the advantage of generating a reduced number of log records at the cost of false dependencies because it does not consider which part of the file is accessed by a process.

4.2 VMA-level

Although an object in memory, such as a file, may be allocated more than one memory region or VMA, a monitor at this level only improves on the object-level if the size of the object forces the operating system to allocate more than one memory region. Our VMA-level monitor can log *read* and *write* accesses to the individual memory regions allocated to the mapped object. We eliminate false dependencies such as when a process writes to a VMA and another process reads from a different VMA allocated to the same object. Our technique to monitor at this level is also based on page protection and page faults monitoring to detect *read* and *write* operations on the memory address space where the file is mapped.

4.3 Page-level

Our page-level monitor starts tracing a memory-mapped file when a process makes an `mmap` system call, The monitor tags the VMA(s) allocated to the process. At this point there is no physical memory page allocation because Linux implements demand paging. The strategy is to generate page faults when the process tries to perform a *read/write* memory operation on the pages of a tagged VMA. Other than page faults resulting from the regular operating system protection mechanism, such as allocating a page upon first time use, address space violation control, and so on, a page fault can also be due to the PTE status bits manipulation implemented in the monitor. When a page fault occurs, the page fault handler checks if the requested page is part of a tagged VMA. If this is the case, and if the memory address is a valid one, a *ticket* is issued. A *ticket* is a combination of process ID, type of memory access operation (*read* or *write*), and the page frame number. Figure 1 shows the implementation of the monitor in the page fault handler.

A *ticket* is valid until one of the following events occurs:

- The CPU scheduler removes the process from the CPU
- The process accesses (either *read* or *write*) a page different from that for which the *ticket* was issued
- The process writes in a page for which a *ticket* was issued for read access only

A *ticket* is not a guarantee of the memory operation taking place at the time of issuing because, even when the

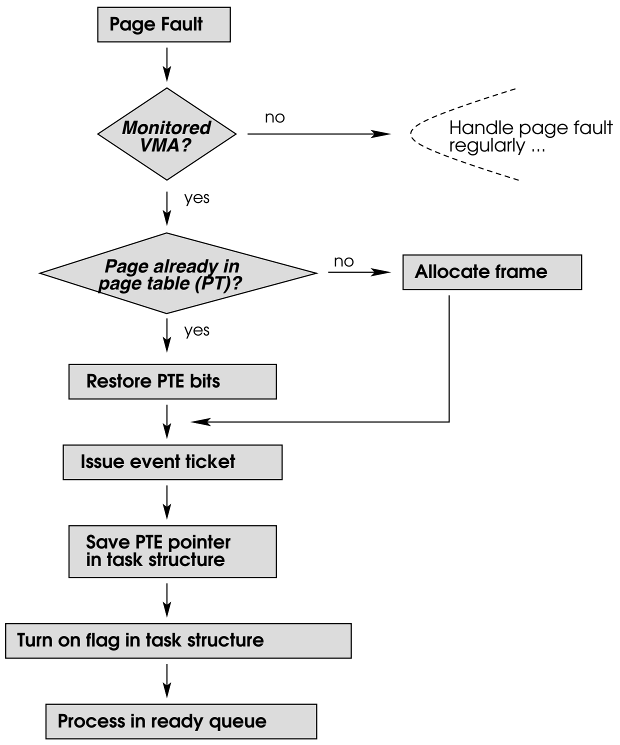
Page Fault
Monitored VMA?
no
Handle page fault regularly ...
yes
Page already in page table (PT)?
no
Allocate frame
yes
Restore PTE bits
Issue event ticket
Save PTE pointer in task structure
Turn on flag in task structure
Process in ready queue

Figure 1: **The page fault handler with page-level monitoring.**

page has been serviced and ready to use, it is not known when the process will resume execution in a multitasking system such as Linux. It has to be stamped in order to confirm the process operated on the faulting page. It is done by the CPU scheduler at the time it allocates CPU to the process to resume execution after the page fault. The stamp is basically the time of effective access and it avoids race conditions such as when a process *P2* resumes execution before *P1* does even when P2's ticket was issued after P1's ticket.

When the CPU scheduler removes a process from execution, it checks whether the process has a monitored page in use so the protection and access bits for that page will be set back off again. This will guarantee the occurrence of a page fault the next time the process tries to access the page and we will be able to log this new memory access event in the future. Figure 2 shows the role of the scheduler in the monitor implementation.

The monitor generates one memory access log record for any number of consecutive *read* operations on the same page as far as they are conducted during the same CPU allocation time slice. This means that the stamped ticket is valid for a specific type of operation on a specific page, and it is revoked when the process is removed from the

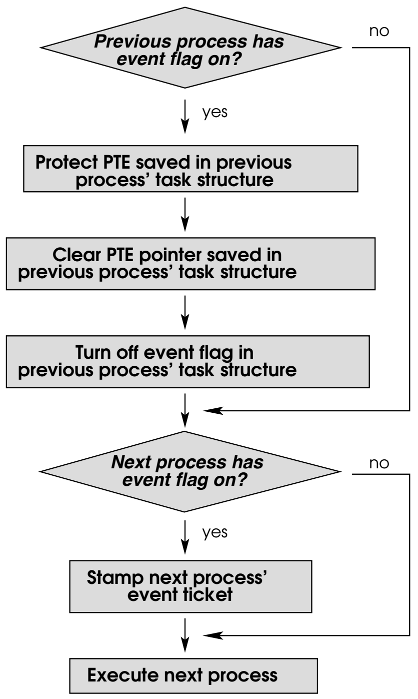
Previous process has event flag on?
no
yes
Protect PTE saved in previous process' task structure
Clear PTE pointer saved in previous process' task structure
Turn off event flag in previous process' task structure
Next process has event flag on?
no
yes
Stamp next process' event ticket
Execute next process

Figure 2: **The CPU scheduler with page-level monitoring.**

CPU. The same applies to consecutive writes. However, for mixed *read* and *write* operations in the same page during the same CPU time slice the following considerations apply:

- *Read followed by write*: The monitor can detect this transition because a page fault will be generated since the PTE will not have the write permissions on. The page fault generated when the process attempted the first *read* issued the corresponding ticket for *read* operations but did not turn the write bit on for the PTE. Therefore, as soon as the process tries to write on the page, a page fault will occur and a new ticket has to be issued.

- *Write followed by read*: This transition cannot be detected by the monitor. The limitation arises because the x86 architecture defines access permissions for memory pages using one bit which takes value 0 when the page is readable and value 1 when it can be written onto. A writable page is, by default, also readable. Once the monitor issues the ticket and sets the protection bit to allow write access, the process will not generate a page fault for a *read* operation. This limitation can be minimized by implementing the monitor at a finer granularity such as minipages, which is part of our future work.

We implemented our prototype in the Linux kernel 2.4.20-8. The monitor runs in kernel space and uses a circular buffer to store records that represent accesses to the file in memory. A daemon reads the buffer and saves the records in permanent storage. There are three types of records:

Map-Event: The request made by the process to open a memory-mapped file. It is generated when the process makes the `mmap` system call. The information in this record is an event number, a process ID, the mapping type (*read/write* and *shared/private*), and the inode number that represents the file.

RW-Event: The *read/write* operation executed by the process. Contains an event number, the Map-Event number that created the mapping, and the page number being accessed.

Unmap-Event: The unmapping of the file as a result of an `munmap` system call or process termination. It contains an event number and the event number of the Map-Event that corresponds to the creation of the mapping of this file.

To test our prototype, we created two processes that mapped the same 100KB data file in memory with *write* permissions and in *shared* mode. We programmed these processes to accept commands such as type of operation (*read* or *write*), number of operations to execute, offsets within the mapped file, and so on, so we could verify the correctness of the logs produced by our monitor. Our page-level monitor produced an accurate sequence of accesses to the memory-mapped file with granularity of a page. The performance impact was measured in terms of added page faults that derived from protecting those memory pages allocated to the memory-mapped file. The penalty was one page fault for any number of consecutive *read* operations to the same page during the same CPU time slice, and one page fault for any number of consecutive *write* operations to the same page during the same CPU time slice.

4.4 Minipage-level

We are currently working on a mechanism to monitor memory access to constituent parts smaller than a page. Our technique consists on instruction replacement to take advantage of debugging exceptions supported in the x86 architecture [17].

5 Related Work

Traditional sources of evidence are network logs and hard disk state. Snort can log network traffic [24]. The Coroners Toolkit (TCT) [10, 7] can recover deleted files and provide information regarding the time a file was created, modified, or accessed. Other forensic tools based on *post-facto* evidence gathering are: EnCase by Guidance Software [15], Forensic Toolkit by Access Data [1], iLook by Internal Revenue Services [18], and SMART by ASR Data [2]. A common constraint of all these forensic analysis tools is that they do not log potential evidence at run-time. Instead, they only count on available *post-mortem* evidence.

Backtracker [21] constructs a graph showing the administrator potential sequences of events that may have influenced the detection point of system compromise. It does so by logging events and system objects at the OS system call level at runtime. Events are identified by monitoring system calls such as *read, write, execv, fork*, and so on. System objects are processes, files, and filenames. A dependency relationship is composed of three parts: a sink object (process reading a file), a source object (file being read), and a time interval where the event took place. The time of each event is very important to establish dependencies between objects. For example, a process who wrote file *X* at time *10* has no dependency on a process that read that file at time *5*. Backtracker achieves its goal of building a graph with potential sequences of events and affected objects from the detection point back to the origin of the attack. However, it does not monitor operations in memory-mapped objects.

Network logs are not sufficient for encrypted communication because the key may not be available, and also they usually provide mostly application-level information. Disk images only keep the final state of the file system but can not provide information of what transpired during the attack. This is a problem when trying to reconstruct the sequence of events that led to the modified objects that triggered the detection point. Implementing in the kernel gives the advantage of being more difficult to disrupt. Kernel implementations can also cope with encryption or obfuscation.

Current operating systems lack built-in mechanisms and tools for comprehensive logging of system events. SNARE [3] is an intrusion reporting and analysis environment which aims to overcome this problem through the implementation of a dynamically loadable kernel module running as a daemon in the Linux operating system. It is capable of providing the forensic analyst with logs for system events such as network connections, reads/writes to files and directories, users and group identity modifications, and changes to programs usage. SNARE is implemented through a loadable kernel module to capture events at the system call level, an audit daemon that defines the type of events and system objects to be audited, and a GUI to present the forensic analyst a high level view of captured events. SNARE does not provide any automatic analysis for sequences of events reconstruction, leaving the analysis task to the investigator after she is presented with the high level view of system events. Similar to Backtracker and Forensix, SNARE sees files as an atomic entity. Shared memory access is not audited.

Forensix [12] improves the accuracy of forensic analysis while reducing the human overhead of performing it. Forensix is based on SNARE [3] and extends its capabilities through three key ideas: Monitoring the execution of the target system at the system call level, providing an application-independent view of all the activity; a secured backed system where the streams of system-call level information are stored through a private interface; and database technology to support high-level querying of the archived logs. Forensix uses the same system call and objects logging technique as Backtracker. It augments potential evidence gathering by logging system call parameters and return values. The main difference with Backtracker is the analysis phase: Forensix relies on database technology to process arbitrary queries to retrieve the sequences of events from the detection point back to the origin of the attack. Similar to Backtracker, Forensix considers a file object as an atomic entity without further granularity. Forensix uses a timing approach to come around the memory access issues discussed above in Backtracker, but still does not log memory load and store operations.

Virtual machine technology is also being used in forensic analysis to provide a framework for the recreation of events. A virtual machine monitor (VMM) is a software implementation that emulates the hardware of a computer system [13, 14]. The operating system running in the virtual machine is called the guest operating system to differentiate it from the operating system running on the bare hardware which is called the host operating system. Implemented in the host OS, ReVirt [8] is an intrusion detection and analysis system that logs every instruction executed in the virtual machine, allowing for later replaying of the virtual machine execution at the instruction-by-instruction level. It is capable of recreating events even when they may have been affected by non determinism. Although ReVirt provides a complete record of the system execution, replaying the virtual machine does not add any analysis for the reconstruction of the sequences of events that led to a system compromise. The logging and replaying mechanism is intended to provide a recreation of the potential attack leaving the analysis to the forensic investigator skills.

Although conceived as an IDS, Livewire [11] provides real-time event logging capabilities for forensic analysis. Implemented in a VMM, it monitors specific events and objects in the guest operating system.

A more recent work presented in [25] improves Backtracker adding fine granularity to events where file objects participate. *Read* and *write* operations in the file system are also monitored at the system call level, with the addition of offset parameters to determine whether a process reads the portion of a file that was previously modified by another process. The result is a reduction in search space, search time, and false dependencies. However, similarly to Backtracker, it does not monitor operations in memory-mapped files.

6 Future Work

We will improve our technique to avoid unnecessary page faults such as when a process *P1* continues to access the same page in a different CPU time slice and no other process accessed that page between *P1's* CPU allocations.

We continue to study hardware support to monitor memory access at a unit smaller than a page, such as a mini-page. A monitor at this level will include a mechanism for temporary replacement of instructions after a page fault. The idea is to exploit the hardware support present in the x86 for debugging using breakpoints through the *int3* instruction, and generate a trap after a *read* or *write* memory operation is executed. The trap will allow to protect the previously used page again, restore the original instruction back to its position, and decrement the program counter (PC) of the process by one. Figure 3 shows a feasible implementation using this technique.

A similar approach for memory access monitoring was presented in the Fault Interpretation (FI) library [9], which can monitor individual read and write accesses to memory through a combination of instruction replacement and page protection. The basic idea in *FI* is to keep a page protected all the time, and unprotect it only for one single access. When an access causes a fault, the page is unprotected and the subsequent instruction

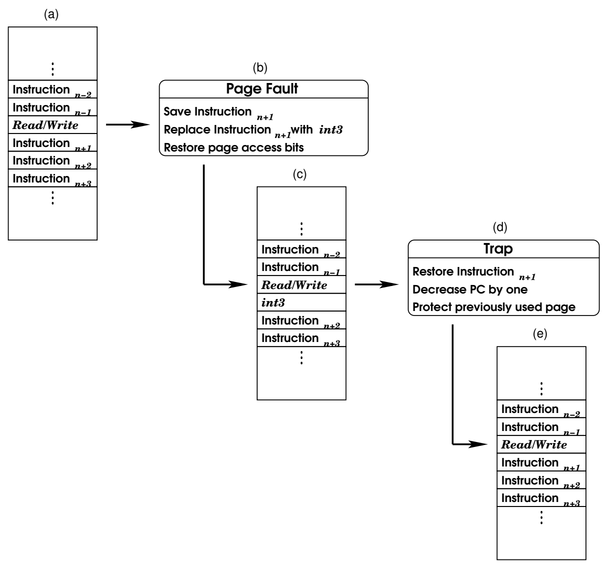

Figure 3: **Instruction Replacement for Minipage-Level Monitoring.** (a) Process attempts to execute a memory *read* or *write* operation and a page fault is generated because the page is protected. (b) The page fault handler saves the subsequent instruction (*instruction*$_{n+1}$), replaces the location with an *int3* instruction, and removes the protection bits so the process can continue execution. (c) Process continues execution at *instruction n* (the previously faulting instruction), but will generate a trap in *instruction*$_{n+1}$ with the *int3* instruction. (d) The trap handler restores *instruction*$_{n+1}$, decreases the program counter to point to *instruction*$_{n+1}$, and protects the page to produce a page fault on the next memory access to the page. (e) Process continues normal execution at *instruction*$_{n+1}$.

is replaced so that, after the access instruction executes, control can return to the library's reprotect block. This way the page is protected again, and the subsequent instruction previously replaced is restored for normal execution. The next *read* or *write* operation will go through the same process and so on. However, *FI* can not guarantee the exact time in which the memory access is actually executed after the page is unprotected because we do not know when the CPU will be allocated to the process to resume its execution in a multitasking operating system.

In addition to file memory-mapping, we are currently working on monitoring memory accesses for other uses of shared memory such as interprocess communication (IPC) through shared memory buffers and threads in a single process.

7 Conclusions

We have presented our work to monitor *read* and *write* operations to memory-mapped files. Our approach provides granularity to gather potential evidence at runtime. It adds accuracy to previous systems that reconstruct sequences of events by reducing the number of false dependencies that can originate when no further monitoring exist for memory-mapped objects. We presented four different levels of granularity to monitor these objects and our runtime monitor prototype demonstrates a page-level monitor for shared memory mapped files provides more detailed information regarding operations in memory. Although monitoring at a smaller unit than that of a page involves losing paging hardware support, instruction replacement was outlined in this paper as our current research path to monitor memory access even at the byte level. When integrated with reconstruction systems, such as Backtracker and Forensix, our monitoring techniques will provide reduction of search space, search time, and false dependencies.

References

[1] Access Data. Forensic Toolkit. http://www.accessdata.com/, February 2005.

[2] ASR Data. SMART. http://www.asrdata.com/SMART/, February 2005.

[3] R. C. Barnett. Catching Intruders with SNARE. http://www.sans.org/rr/audittech/Ryan_Barnett_AT.pdf, April 2003. SANS Audit Controls that Work.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, 2nd edition, February 2003.

[5] B. Carrier. Defining Digital Forensic Examinations and Analysis Tools Using Abstraction Layers. *International Journal of Digital Evidence*, 1(4), Winter 2003.

[6] B. Carrier and E. Spafford. Defining Event Reconstruction of Digital Crime Scenes. *Journal of Forensic Science*, 49(6), November 2004.

[7] D. Darmer and W. Venema. *Forensic Discovery*. Addison-Wesley Professional Computing Series. Addison Wesley Professional, 1st edition, December 30, 2004.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M.Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[9] D. R. Edelson. Fault Interpretation: Fine-Grain Monitoring of Page Accesses. In *Proceedings of the 1993 Usenix Winter Conference*, pages 395–404. USENIX, January 1993.

[10] D. Farmer and W. Venema. The Coroner's Toolkit (TCT). http://www.porcupine.org/forensics/tct.html.

[11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

[12] A. Goel, M. Shea, S. Ahuja, W.-C. Feng, W.-C. Feng, D. Maier, and J. Walpole. Forensix: A Robust, High-Performance Reconstruction System. In *Forensix: A Robust, High-Performance Reconstruction System*, 2003. [Poster Session].

[13] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, 1974.

[14] M. Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall PTR, 1^{st} edition, April 29, 2004.

[15] Guidance Software. EnCase Enterprise Edition Detailed Product Description.
https://www.guidancesoftware.com/, July 2004.

[16] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[17] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual - Volume 3 - System Programming Guide. Order Number: 253668-015, April 2005.

[18] Internal Revenue Services. iLook Investigator Toolsets. http://www.ilook-forensics.org/, February 2005.

[19] A. Itzkovitz and A. Schuster. MultiView and Millipage Fine-Grain Sharing in Page-Based DSMs. In *Proceedings of The 3^{rd} Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 215–228, New Orleans, LA, USA, February 1999.

[20] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of 1994 ACM Conference on Computer and Communication Security (CCS)*, pages 18–29, Fairfax, VA, USA, 1994. ACM Press.

[21] S. T. King and P. M. Chin. Backtracking Intrusions. In *Proceedings of the 19^{th} ACM Symposium on Operating Systems Principles*, Session: Making operating systems more robust, pages 223–236, Bolton Landing, NY, USA, 2003. ACM Press.

[22] G. Palmer. A Road Map for Digital Forensic Research. Technical Report DTR-T0010-01, Digital Forensic Research Workshop (DFRWS), November 2001.

[23] M. Reith, C. Carr, and G. Gunsch. An Examination of Digital Forensics Models. *International Journal of Digital Evidence*, Fall 2002.

[24] M. Roesch. Snort-Lightweight Intrusion Detection for Networks. In *Proceedings of the 13^{th} USENIX Conference on System Administration (LISA '99)*, pages 229–238, Seattle, WA, USA, 1999. USENIX Association.

[25] S. Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions Using Improved Backtracking. In *Third IEEE International Workshop on Information Assurance (IWIA'05)*, College Park, Maryland, USA, 2005. IEEE Association.