Forenscope: A Framework for Live Forensics

Ellick Chan† emchan@illinois.edu
Shivaram Venkataraman† venkata4@illinois.edu
Francis David* francis.david@microsoft.com
Amey Chaugule† achaugu2@illinois.edu
Roy Campbell† rhc@illinois.edu

ABSTRACT

Current post-mortem cyber-forensic techniques may cause significant disruption to the evidence gathering process by breaking active network connections and unmounting encrypted disks. Although newer live forensic analysis tools can preserve active state, they may taint evidence by leaving footprints in memory. To help address these concerns we present Forenscope, a framework that allows an investigator to examine the state of an active system without the effects of taint or forensic blurriness caused by analyzing a running system. We show how Forenscope can fit into accepted workflows to improve the evidence gathering process.

Forenscope preserves the state of the running system and allows running processes, open files, encrypted filesystems and open network sockets to persist during the analysis process. Forenscope has been tested on live systems to show that it does not operationally disrupt critical processes and that it can perform an analysis in less than 15 seconds while using only 125 KB of memory. We show that Forenscope can detect stealth rootkits, neutralize threats and expedite the investigation process by finding evidence in memory.

Keywords: forensics, introspection, memory remanence

1. INTRODUCTION

Current forensic tools are limited by their inability to preserve the hardware and software state of a system during investigation. Post-mortem analysis tools require the investigator to shut down the machine to inspect the contents of the disk and identify artifacts of interest. This process breaks network connections and unmounts encrypted disks causing significant loss of potential evidence and possible disruption of critical systems. In contrast, live forensic tools can allow an investigator to inspect the state of a running machine without disruption. However existing tools can overwrite evidence present in memory or alter the contents of the disk causing forensic *taint* which lowers the *integrity* of the evidence. Furthermore, taking a snapshot of the system can result in a phenomena known as forensic *blurriness* [26] where an inconsistent snapshot is captured because the system is running while it is being observed. Forensic blurriness affects the *fidelity* and quantity of evidence acquired and can cast doubt on the validity of the analysis, making the courts more reluctant to accept such evidence [4].

Experts at the SANS institute and DOJ are starting to recognize the importance of volatile memory as a source of evidence to help combat cybercrime [1, 3]. In response, the SANS institute recently published a report on volatile memory analysis [7]. To help address the limitations of existing volatile memory analysis tools we present Forenscope, a framework for live forensics, that can capture, analyze and explore the state of a computer without disrupting the system or tainting important evidence. Section 2 shows how Forenscope can fit into accepted workflows to enhance the evidence gathering process.

Forenscope leverages DRAM memory remanence to preserve the running operating system across a "state-preserving reboot"(Section 3) which recovers the existing OS without having to go through the full boot-up process. This process enables Forenscope to gain complete control over the system and perform taint-free forensic analysis using well grounded introspection techniques [22]. Finally, Forenscope resumes the existing OS, preserving active network connections and disk encryption sessions causing minimal service interruption in the process. Forenscope captures the contents of system memory to a removable USB device and activates a software write blocker to inhibit modifications to the disk. To maintain fidelity, it operates exclusively in 125 KB of unused legacy conventional memory and does not taint the contents of extended memory. Since Forenscope preserves the state of a running machine, it is suitable for use in production and critical infrastructure environments. We have thoroughly tested and evaluated Forenscope on an SEL-1102, a power substation industrial computer, and an IBM desktop workstation. The machines were able to perform their duties under a variety of test conditions with minimal interruption and running Forenscope did not cause any network applications to time out or fail. Our current implementation is based on Linux 2.6, although the technique is also applicable to other major operating systems.

We have implemented several modules that can check for the presence of malware, detect open network sockets and locate evidence in memory such as rootkit modifications to help the investigator identify suspicious activity.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

†University of Illinois, *Microsoft

The contributions of this work include:

1. An extensible software framework for high-fidelity live forensics conforming to the best practices of a legal framework of evidence.
2. Efficient techniques to gather, snapshot and explore a system without bringing it down.
3. Implementation and evaluation on several machines including a standard industrial machine and against several anti forensics rootkits.

This paper is organized as follows: Section 2 introduces cyber-forensics followed by Section 3 which describes the design of Forenscope. We evaluate the effectiveness of Forenscope in Section 4. Section 5 discusses forensics issues, Section 6 surveys related work and Section 7 concludes.

2. BACKGROUND

To provide an overview of the accepted legal framework of evidence collection currently in place, we summarize the workflow from the CERT guide on FBI investigation [10]:

1. Preserve the state of the computer by creating a backup copy of logs and files left by the intruder.
2. If the incident is in progress, log activity.
3. Document the losses suffered by your organization.
4. Contact law enforcement.

While the steps executed are similar for various cases, there are special requirements for each case. For instance, in criminal investigation, integrity and fidelity of the data is paramount. As evidence presented in court must be as accurate as possible, special steps must be taken to ensure fidelity. For incident response, the goal is to detect and react to security breaches while minimizing the intrusiveness of the process. In some critical systems it is impractical to interrupt the system to perform forensic analysis of a potential breach and service level agreements (SLAs) may impose financial penalties for downtime. The cases chosen above are example of evidentiary requirements but a more thorough analysis is beyond the scope of this paper. To preserve the fidelity of the original evidence, many forensic workflows capture a pristine image of the evidence and draw conclusions based on analysis of the copy. Conventional post-mortem forensic workflows perform this task by physically shutting down a computer and copying the contents of the hard drive for subsequent analysis. On the other hand, live forensics are often desired for step 2 because they provide access to networked resources such as active SSH and VPN sessions, remote desktop connections, IM clients and file transfers. However even state-of-the-art solutions often cannot image a system with high fidelity and frequently introduce taint in the process. In summary, existing tools require the investigator to make a tradeoff between increased fidelity through post mortem analysis or the potential to collect important volatile information using live forensic tools at the cost of tainting evidence.

One of the key issues in collecting volatile information is that various forms of data such as CPU registers, memory, disk and network connections have different lifetimes. To maximize evidence preservation, RFC 3227 [8] outlines the *order of volatility* of these resources and dictates the order in which evidence should be collected for investigation. Commercial products currently used by forensic experts for incident response such as Encase, Helix, FTK Imager and Memoryze [1] etc, do not capture all forms of data. A comparison of these products is presented in Table 1. Scalpel and Sleuth kit are solely designed for disk analysis while other tools such as Encase, Helix and FTK include some level of memory capture and analysis capability. Memoryze is the only tool listed in the table that performs volatile memory analysis. Some tools such as Helix, FTK and Memoryze can list the state of open network sockets, but the underlying network connections are not preserved during the analysis process. All live forensic tools listed in this table rely on the integrity of the running kernel. Compromised systems may provide inaccurate information. Evidence preservation and minimizing forensic intrusiveness are hard problems that haven't been adequately addressed in the literature.

In contrast, Forenscope was built to comply with steps 1 and 2 where it maximizes the preservation of evidence and avoids disruption of ongoing activities to allow the capture of high fidelity evidence. As a result, we believe that Forenscope may be more broadly applicable to various scenarios which require live forensics such as incident response and criminal investigation. For incident response, we recognize that the integrity of the machine may be violated by malware and our solutions have been designed to address this scenario. For criminal investigation, we presume that the machine may have various security mechanisms implemented such as encrypted disks coupled with authentication mechanisms such as logon screens and screensaver locks.

3. DESIGN

Forenscope utilizes the principle of introspection to provide a consistent analysis environment free of taint and blurriness which we term as the *golden state*. In this state, the system is essentially quiescent and queries can be made to analyze the system. As a result, analysis modules can access in-memory data structures introspectively. The investigator activates forenscope by forcing a reset where the state of the machine is preserved by memory remanence in the DRAM chips. Then, the investigator boots off the Forenscope media which performs forensic analysis on the latent state of the system and restores the functionality of the system for further live analysis. Forenscope is designed to work around security mechanisms by interposing a lightweight analysis platform beneath the operating system. For example, in incident response, the machine may be controlled by malicious software and the operating system cannot be trusted. The observation capabilities afforded by Forenscope offer additional visibility in these scenarios.

3.1 Taint and Blurriness

Taint and blurriness are concepts related to the use of forensic tools. Taint is a measurement of change in the system induced by the use of a forensic tool and it may be present both in memory and on disk. In this section, we only consider the in-memory portion because BitBlocker (Section 3.6) eliminates disk taint by blocking writes. Blurriness refers to the inconsistency of a memory snapshot taken while a system is running.

[1]Encase: www.encase.com,
Helix: www.e-fense.com,
FTK Imager: www.accessdata.com,
Memoryze: www.mandiant.com/software/memoryze.htm

Table 1: Comparison of Forenscope with existing forensic tools

Evidence	Registers	Memory	Network	Processes	Disk	Encryption
RFC 3227 Reqs	Nanosecs	Seconds	Minutes	Minutes	Hours	Hours
Encase	×	✓[a]	×	×	✓	×
Helix	×	✓[a]	✓[b]	✓	✓	×
FTK	×	✓[a]	✓	✓	✓	✓
Scalpel	×	×	×	×	✓	×
Memoryze	×	✓[a]	✓[b]	✓	×	×
Sleuth kit	×	×	×	×	✓	×
Forenscope	✓	✓	✓	✓	✓	✓

[a] Subject to forensic blurriness
[b] Connection is recorded but not persisted

Table 2: Definitions

Quantity	Description
Snapshot $\overline{S_t}$	Contents of memory at time t
Natural drift δ_v	Change in the system state over time v
Snapshot $\hat{S_v}$	Contents of captured memory snapshot with v being the time taken to capture the snapshot
Taint f	f is defined as the memory taint caused by the forensic introspection agent

To quantify the relationship between taint and blurriness, let $\overline{S_t}$ be the contents of memory at any given instant of time t. The state of a system changes over a period of time due to the natural course of running processes and we define this as the natural drift of the system, δ. When a traditional live forensic tool attempts to take a snapshot of the system, there is a difference between what is captured, $\hat{S_v}$ and the true snapshot $\overline{S_t}$, where v represents the time taken to capture the snapshot. There are two reasons for this difference: the first being δ_v the natural drift over the time period when the snapshot was being acquired (v) and the second due to the footprint f of the forensic tool. We define the former as the blurriness of the snapshot and the latter quantity to be the taint caused by the forensic tool. Table 2 captures these definitions in a concise form. In general, there are two ways to obtain a snapshot of the machine's state: active techniques and passive techniques. Active techniques involve the use of an agent on the machine which may leave a footprint. Passive techniques operate outside the domain of the machine and do not affect its operation, one such example is VM introspection. When a passive acquisition tool is used, the relationship $\hat{S_v} = \overline{S_t} + \delta_v$ indicates that the approximate snapshot differs from the true snapshot due to the blurriness δ_v. In contrast, when an active forensic tool is used, $\hat{S_v} = \overline{S_t} + f + \delta_v$, where f represents taint and δ_v represents blurriness. Collectively, these quantities are a measure of the error in the snapshot acquisition process. Taint can result from the direct action of forensic tools or indirect effects induced in the system through the use of these tools. We call the former first-order taint, f', and the latter second-order taint, f''. First-order taint can result from loading a forensic tool into memory and second-order taint can result from processes such as file buffering due to the effects of a forensic tool writing a file.

3.2 Memory Remanence

Modern memory chips are composed of capacitors which store binary values using charge states. Over time, these capacitors leak charge and must be refreshed periodically. To save power, these chips are designed to retain their values as long as possible, especially in mobile devices such as laptops and cell phones. Contrary to common belief, the act of rebooting or shutting down a computer often does not completely clear the contents of memory. Link and May [21] were the first to show that current memory technology exhibited remanence properties back in 1979. More recently, Gutmann [18] elaborated on the properties of DRAM memory remanence. Halderman et al. [19] recently showed that these chips can retain their contents for tens of seconds at room temperature and the contents can persist for several minutes when the RAM chips are cooled to slow the natural rate of bit decay. Forenscope utilizes memory remanence properties to preserve the full system state to allow recovery to a point where introspection can be performed. We refer the reader to [11, 19] for a more detailed analysis of memory remanence.

3.3 Activation

Forenscope currently supports two methods of activation. The first is based on a watchdog timer reset and the second is through a forced reboot. For incident response, a watchdog timer may be used to activate Forenscope periodically to audit the machine's state and check for the presence of stealth malware. Watchdog timers are used in embedded systems to detect erroneous conditions such as machine lockups. These timers contain a count down clock which must be refreshed periodically. If the system crashes, the watchdog software will fail to refresh the clock. Once the clock counts down to zero, the watchdog timer will issue a warm hardware reset signal to the machine causing it to reboot in the hopes that the operating system will recover from the erroneous condition upon a fresh start. On our test machine, the built-in watchdog timer is programmable via a serial port interface and the contents of DRAM memory are not cleared after a reboot initiated by the watchdog timer reset signal.

On the other hand, a forensic investigator may encounter a machine that is locked by a screensaver or login screen and in this situation, Forenscope can be activated by forcing a reboot. Some operating systems such as Linux and Windows can be configured to reboot or produce a crash dump by pressing a hotkey. These key sequences are often used for

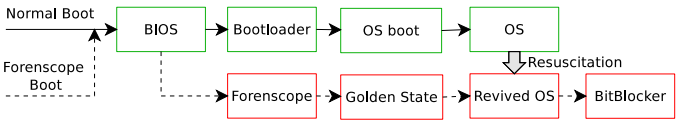

Figure 1: Forenscope vs normal boot paths

debugging and are enabled by default in many Linux distributions. In Linux, the *alt-sysrq-b* hotkey sequence forces an immediate reboot. If these debug keys are disabled, then a reset may be forced by activating the hardware reset switch. Forenscope supports multiple modes of operation for versatility. After the machine has been rebooted forcefully, the Forenscope kernel is selected from the boot loader menu instead of the incumbent operating system.

3.4 Forenscope framework

Instead of booting afresh, Forenscope alters the boot control flow to perform its analysis. Figure 1 illustrates this process. After the machine restarts, it boots off a CD or USB stick with the Forenscope media. The machine then enters the *golden state* monitor mode which suspends execution and provides a clean external view of the machine state. To explain how the monitor works, we first describe the operating states of the x86 architecture. When a traditional PC boots, the processor starts in *real mode* and executes the BIOS. The BIOS then loads the bootloader which in turn loads the operating system. During the boot sequence, the operating system first enables `protected mode` to access memory above the 1 MB mark and then sets up page tables to enable virtual memory to bootstrap the OS. Forenscope interposes on this boot sequence and first establishes a bootstrap environment residing in the lower 640 KB rung of legacy conventional memory and then it reconstructs the state of the running machine. Forenscope has full control of the machine and its view is untainted by any configuration settings from the incumbent operating system because it uses a trustworthy private set of page tables; thus rootkits and malware which have infected the machine cannot interfere with operations in this state. Next, Forenscope obtains forensically-accurate memory dumps of the system and runs various kinds of analyses. For integrity, Forenscope does not rely on any services from the underlying operating system. Instead, it makes direct calls to the system's BIOS to read and write to the disk. Therefore, Forenscope is resistant to malware that impedes the correct operation of hardware devices. The initial forensic analysis modules are executed in this state and then Forenscope restores the operation of the incumbent operating system.

3.5 Reviving the Operating system

To revive the incumbent operating system, Forenscope needs to restore the hardware and software state of the system to "undo" the effects of the reboot. Hardware devices are reset by the BIOS as part of the boot process. Some of these devices must be reconfigured before the incumbent operating system is restored because they were used by Forenscope or the BIOS during initialization. To do so, Forenscope first re-initializes core devices such as the hard drive and interrupt controller and then assumes full control of these devices for operation in its clean environment. Before resuming the operating system, Forenscope scans the PCI bus and gathers a list of hardware devices. Each hardware device is matched against an internal database and if an entry is found, Forenscope calls its own reinitialization function for the particular hardware device. If no reinitialization function is found, Forenscope looks up the device class and calls the operating system's generic recovery function for that device class. Many devices such as network cards and disk drives have facilities for handling errant conditions on buggy hardware. These devices typically have a *timeout recovery* function which can revive the hardware device in the event that it stops responding. We have found that calling these recovery functions is usually sufficient to recover most hardware devices. In Linux, 86 out of the 121 (71%) PCI network drivers implement this interface and all IDE device drivers support a complete device reset. For instance, the IBM uses an Intel Pro/100 card and the SEL-1102 uses a built-in AMD PCnet/32 chip. On both these machines Forenscope relies on calling the `tx_timeout` function to revive the network. We use a two-stage process to restore the operating system environment. The first stage reconstructs the processor state where the values of registers are extracted and altered to roll back the effects of the restart and the second stage runs forensic analysis modules. Our algorithm scans the active kernel stack and symbol information from the kernel for call chain information. Forenscope uses this information to reconstruct the processor's state. In the `alt-sysrq-b` case, the interrupt handler calls the keyboard handler which in turn invokes the emergency sysrq-handler. The processor's register state is saved on the stack and restored by using state recovery algorithms from [11, 13]. If the `alt-sysrq-b` hotkey is disabled, Forenscope supports an alternate method of activation based on pressing a physical reset switch. In this case, Forenscope assumes that the system is under light load and that the processor spends most of its time in the kernel's idle loop. In this loop, most kernels repeatedly call the x86 `HLT` instruction to put the processor to sleep. Since the register values at this point are predictable, Forenscope restores the instruction pointer, `EIP`, to point to the idle loop itself and other registers accordingly. Once the state has been reconstructed, Forenscope reloads the processor with this information and enables virtual memory.

3.6 Modules

We have developed a number of modules to aid in forensic analysis. These modules, shown in Figure 2, run in groups where stage 1 modules run in the golden state to collect pristine information while stage 2 modules rely on OS services to provide a shell and block disk writes. Finally, stage 3 resumes the original operating environment.

Scribe: Scribe collects basic investigation information such as the time, date, list of PCI devices, processor serial number and other hardware features. These details are stored as evidence to identify the source of a snapshot.

Cloner: Cloner is a memory dump forensic tool that is able to capture a high-fidelity image of volatile memory contents to an external capture device. Existing techniques for creating physical memory dumps are limited by their reliance on system resources which are vulnerable to deception. Cloner works around forensic blurriness issues and rootkit cloaking by running in stage 1 before control is returned to the original host OS. In the golden state, the system uses `protected mode` to access memory directly through Forenscope's safe memory space. Using this technique, Cloner accesses memory directly without relying on services from the incumbent operating system or its page tables. To dump the contents of memory, Cloner writes to disk directly using BIOS services instead of using an OS disk driver. This channel avoids a potentially booby-trapped or corrupted operating system disk driver and ensures that the written data has better forensic integrity. Most BIOS firmware supports read/write access

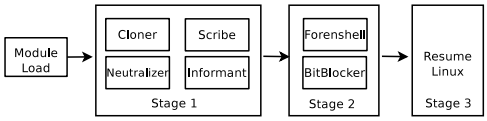

Figure 2: Forenscope modules

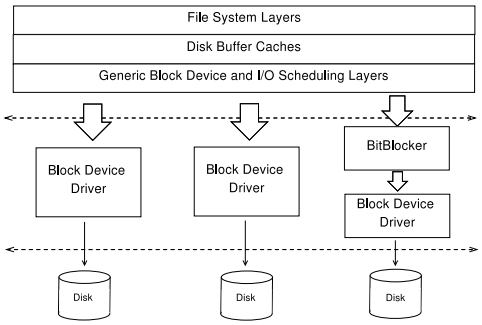

Figure 3: File system architecture

to USB flash drives and hard disks. Another reason to use the BIOS for dumping is that it minimizes the memory footprint of Forenscope and reduces dependencies on drivers for various USB and SATA chipsets. Once cloner captures a clean memory dump, the investigator can run other modules tools that may alter the contents of memory without worry of tainting the evidence.

Informant: Informant checks for suspicious signs in the system that may indicate tampering by identifying the presence of alterations caused by malware. In order to extract clean copies of the program code and static structures such as the system call table, Forenscope must have access to a copy of the `vmlinux` kernel file which is scanned to locate global kernel variables and the location of various functions. Most Linux distributions provide this information. Read-only program code and data structures are checked against this information to ensure that they have not been altered or misconfigured. Such alterations have the potential to hinder the investigation process and Informant helps to assess the integrity of a machine before further analysis is attempted. After Informant verifies the system, it also records other useful information such as the contents of the kernel *dmesg* log, running processes, open files and open network sockets. This information can help expedite the investigation process.

Neutralizer: Neutralizer inoculates against anti-forensic software by detecting and repairing alterations in binary code and key system data structures such as the system call table. These structures can be repaired by restoring them with clean copies extracted from the original sources. Since many rootkits rely on alteration techniques, Neutralizer can recover from the effects of common forms of corruption. Presently, Neutralizer is unable to recover from corruption or alteration of dynamic data structures. Neutralizer also suppresses certain security services such as the screensaver, keyboard lock and potential malware or anti-forensic tools by terminating them. To terminate processes, neutralizer sends a `SIGKILL` signal instead of a `SIGTERM` signal so that there is no opportunity to ignore the signal. Customized signals can be sent to each target process. For some system services that respawn, terminating them is ineffective, so forcefully changing the process state to zombie (Z) or uninterruptible disk sleep (D) is desired instead of killing the application directly. An alternative would be to send the `SIGSEGV` signal to certain applications to mimic the effects of a crash. Neutralizer selects processes to kill based on the analysis mode. For incident response on server machines, a white list approach is used to terminate processes that do not belong to the set of core services. This policy prevents running unauthorized applications that may cause harm to the system. For investigation, Neutralizer takes a black list approach and kills off known malicious processes.

Table 3: Correctness assessment

Application	Results
Idle system	System is correctly recovered over 100 times.
SSH	SSH recovers, protocol handles lost packets.
PPTP VPN	VPN recovers, queued messages are delivered.
AES pipe	File encryption continues.
Netcat	File transfers correctly without checksum errors.
DM-crypt	Mounted filesystem remains accessible.

ForenShell: ForenShell is a special superuser *bash* shell that allows interactive exploration of a system by using standard tools. When coupled with BitBlocker(below), ForenShell provides a safe environment to perform customized live analyses. In this mode, Forenshell becomes non-persistent and it does not taint the contents of storage devices. Once ForenShell is started, traditional tools such as Tripwire or Encase may be run directly for further analysis. To provide an audit log of the investigator's activities, ForenShell provides a built-in keylogger that writes directly to the evidence collection medium without tainting the disk. Forenscope launches the superuser shell on a virtual console by directly spawning it from a privileged kernel thread. ForenShell runs as the last analysis module after Informant and Neutralizer have been executed. At this point, the system has already been scanned for malware and anti-forensic software. If Neutralizer is unable to clean an infection, it displays a message informing the investigator that the output of ForenShell may be unreliable due to possible system corruption.

BitBlocker: BitBlocker is a configurable software-based write blocker that inhibits writing to a given set of storage devices to avoid tainting the contents of persistent media. Since actions performed by ForenShell during exploration can inadvertently leave undesired tracks, BitBlocker helps to provide a safe non-persistent analysis environment that emulates disk writes without physically altering the contents of the media. Because BitBlocker modifies the contents of memory, it executes after Cloner has captured a clean copy of memory.

Simply re-mounting a disk in read-only mode to prevent writing may cause some applications to fail because they may need to create temporary files and expect open files to remain writable. Typically, when an application creates or writes files, the changes are not immediately flushed to disk and they are held in the disk's buffer cache until the system can flush the changes. The buffer cache manages intermediate disk operations and services subsequent read requests with pending writes from the disk buffer when possible. BitBlocker mimics the expected file semantics of the original system by reconfiguring the kernel's disk buffer cache layer to hold all writes instead of flushing them to disk. This approach works on any type of file system because it operates directly on the disk buffer which is one layer below the file system. BitBlocker's design is similar to that of some Linux-based RAM disk systems [5] which cleverly use the disk buffer as a storage system by configuring the storage device with a null backing store instead of using a physical disk. Each time a disk write is issued, barring a *sync* opera-

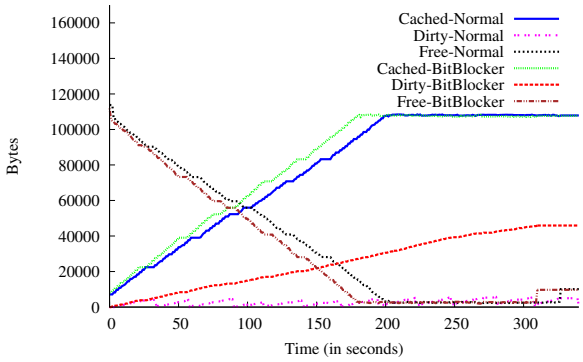

Figure 4: BitBlocker memory usage

tion, the operating system's disk buffer subsystem holds the request in the buffer until a certain write threshold or timeout is reached. In Linux, a system daemon called *pdflush* handles flushing buffered writes to disk. To prevent flushing to the disk, BitBlocker reconfigures the write threshold of the disk to inhibit buffer flushing, disables *pdflush* and hooks the *sync, sync_file_range, fsync, bdflush* and *umount* system calls with a write monitor wrapper. Figure 3 shows the architectural diagram of the Linux filesystem layer and where BitBlocker intercepts disk write operations. Although BitBlocker inserts hooks into the operating system, it does not interfere with the operations of Informant and Neutralizer because those modules are run before BitBlocker and they operate on a clean copy of memory. The hooks and techniques used by BitBlocker are common to Linux 2.6.x kernels and they are robust to changes in the kernel version. Similar techniques are possible for other operating systems.

4. RESULTS AND EVALUATION

We evaluate Forenscope as a forensic tool by measuring five characteristics: correctness, performance, downtime, fidelity and effectiveness against malware.

Hardware and Software Setup: To demonstrate functionality, we tested and evaluated the performance of Forenscope on two machines: a Schweitzer 1102 industrial computer and an IBM Intellistation M Pro. The SEL-1102 used in our experiments is a rugged computer designed for power system substation use and it is equipped with 512 MB of DRAM and a 4 GB compact flash card mounted in the first drive slot as the system disk. The SEL-1102 can operate in temperatures ranging from -40 to +75 degrees Celsius. The IBM Intellistation M Pro is a standard desktop workstation equipped with 1 GB of DRAM. For some tests, we opted to use a QEMU-based virtual machine system to precisely measure timing and taint. Forenscope and the modules that we developed were tested on the Linux 2.6 kernel. Although Forenscope was originally built to target Linux, we plan to expand this work to other systems.

Correctness: To show that Forenscope is robust, we tested it against a collection of applications listed in Table 3. In each case, after rebooting the machine forcefully, Forenscope recovered the operating state, took control and ran successfully without breaking the semantics of the application. As a basic sanity test, Forenscope was able to revive an idle system with no load. We chose a mix of applications to show that a wide range of hardware, software and network applications are compatible. Running SSH, PPTP and Netcat showed that network connections persist. Further

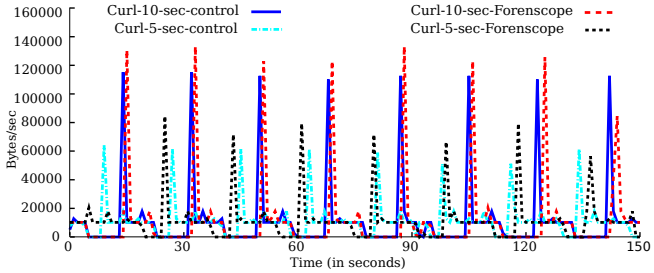

Figure 5: HTTP data transfer rate comparison

testing using DM-crypt and AES pipe showed that security programs continue to operate properly. A more thorough evaluation of the correctness can be found in [11]. To evaluate the correctness of BitBlocker, we ran it on the IBM and on a QEMU system emulator. Using the emulator allowed us to verify integrity by checksumming the contents of the virtual disk. Our test cases include using the *dd* utility to fill up the disk, then issuing a *sync* command and unmounting the disk. Other cases tested include copying large files and compiling programs consisting of hundreds of files. In each case, BitBlocker worked correctly and no writes were issued to the physical disk. After the test completed, we confirmed that the contents of the disk were unchanged by comparing hashes of the contents against the original contents.

Performance: In terms of performance, BitBlocker made disk operations appear to be faster because no data is flushed to the physical disk from the disk buffer. A write of a 128 MB file took 32.78 s without BitBlocker and 3.71 s with BitBlocker. The number of dirty disk buffers consumed increases proportionately with the size of the files written. Since BitBlocker inhibits flushing to disk, running out of file buffers can create a condition where the filesystem fills up and reports a write error. To measure these effects on the system, we collected buffer cache usage information once a second in several key applications: creating a compressed archive with *tar-bzip2*, downloading a file using *wget* and compiling the software package *busybox*. Figure 4 shows the utilization of dirty file buffers over time for the tar-gzip case. Wget and busybox compilation have similar results. In the graphs, we report statistics from `/proc/meminfo` such as `cached`, `dirty` and `free`. According to the documentation for `/proc`, `cached` in Linux represents the amount of data in the page cache which includes cached data from read-only files as well as write buffers. `Dirty` represents items that need to be committed to the disk and free represents free memory. From our observations, `dirty` is generally very low in the normal case because the kernel commits write buffers periodically. However, in BitBlocker, `dirty` grows steadily because the data cannot be committed back to the disk. To estimate the amount of memory required to run BitBlocker, our experiments show that in many scenarios, even 128 MB of free memory is sufficient for BitBlocker to operate. Our experiments show that BitBlocker is robust even when the system runs low in memory. At 200 seconds, the physical memory of the machine fills up and the tar-bz2 process stops because the disk is "full." The system does not crash and other apps continue to run as long as they do not write to the disk. On a typical system with 2 GB of memory, BitBlocker should be able to maintain disk writeability for a much longer period of time.

Table 4: Taint measurement (pages)

Description (32,768)	Conventional Memory	Extended Memory
Forenscope	41 (0.125%)	0(0%)
dd	0 (0%)	7100 (21.66%)
dd to FS mounted with sync flag	0 (0%)	7027 (21.44%)
dd with O_DIRECT	0 (0%)	480 (1.46%)

Downtime: As discussed earlier, one important metric for evaluating a forensic tool is the amount of downtime incurred during use. To show that Forenscope minimally disrupts the operation of critical systems, we measured the amount of time required to activate the system. Forenscope, without Cloner, executed in 15.1 s using the reboot method on the SEL-1102 and in 9.8 s on the IBM Intellistation while the watchdog method took 15.2 s to execute on the SEL-1102. The majority of the downtime is due to the BIOS bootup sequence and this downtime can be reduced on some machines. Many network protocols and systems can handle this brief interruption gracefully without causing significant problems. We tested this functionality by verifying that VPN, SSH and web browser sessions continue to work without timing out despite the interruption. Many of these protocols have a timeout tolerance that is sufficiently long to avoid disconnections while Forenscope is operating and TCP is designed to retransmit lost packets during this short interruption. To measure the disruption to network applications caused by running Forenscope continuously over a period of time, we ran a test within a virtualized environment to mimic the brief reboot cycle used by the analysis process. The test measures the instantaneous speed of an HTTP file transfer between a server and a client machine. While the file transfer is in session, we periodically interrupt the transfer by forcibly restarting the machine and subsequently reviving it using Forenscope. Each time the system is interrupted, the server process is suspended while the machine reboots. The process is then resumed once Forenscope is done running. As a baseline, we created a control experiment where the server process is periodically suspended and resumed by a shell script acting as a governor to limit the rate at which the server operates. This script sends the `SIGSTOP` signal to suspend the server process, waits a few seconds to emulate the time required for the bootup process and then sends a `SIGCONT` signal to resume operation. In each experiment, a `curl` client fetches a 1 MB file from a `thttpd` server at a rate of 10 KB/s. We chose these parameters to illustrate how a streaming application or low-bandwidth application such as a logger may behave. During this download process, the server was rebooted once every 20 seconds and we measured the instantaneous bandwidth with a bootup delay of 5 and 10 seconds to observe the effects of various bootup times. We observed that the bandwidth drops to zero while the system boots and the download resumes promptly after the reboot. No TCP connections were broken during the experiment and the checksum of the downloaded file matched that of the original file on the server. A graph of the instantaneous bandwidth vs time is plotted in Figure 5. We compared the results of our test against the control experiment and observed that the behavior was very similar. Thus we believe that running Forenscope can be considered as safe as suspending and resuming the process. During the experiment we noticed that the bandwidth spiked immediately after the machine recovered and attribute this behavior to the internal 2-second periodic timer used by *thttpd* to adjust the rate limiting throttle table.

Taint and Blurriness: We evaluated the taint in a snapshot saved by Forenscope using a snapshot captured by `dd` as the baseline. In an experimental setup running with 128 MB of memory, we collected an accurate snapshot $\overline{S_t}$ of the physical memory using QEMU and compared that with a snapshot $\hat{S_v}$ obtained from each forensic tool. The number of altered pages for each of the configurations is presented in Table 4. We observe that since Forenscope is loaded in conventional memory, the only pages which differ are found in the lower 640 KB of memory. Our experiments show that Forenscope is far better than `dd` because we observed no difference in the extended memory between the snapshot taken by Forenscope and the baseline snapshot. It should be noted that as the machine is suspended in the golden state when running Forenscope, there is no blurriness associated with the snapshot taken by Forenscope. For *dd*, we measured the taint when using a file system mounted with and without the *sync* option. The number of pages affected remains almost the same in both cases and we observed that the majority of second-order taint was due to the operating system filling the page-cache buffer while writing the snapshot. To evaluate how much taint was induced due to buffering, we ran experiments in which *dd* was configured to write directly to disk, skipping any page-cache buffers by using the `O_DIRECT` flag. The results show that the taint was much lower than the earlier experiment, but still greater than the taint caused by using Forenscope. In order to estimate the amount of blurriness caused when tools like *dd* are used, we measured the natural drift over time of some typical configurations. We collected and compared memory dumps from Ubuntu 8.04 and Windows Vista with 512 MB of memory in a virtual machine environment hosted in `QEMU`. In each case, we snapshot the physical memory of the virtual machine and calculate the number of pages that differ from the initial image over a period of time. The snapshots were sampled using a *tilted time frame* to capture the steady state behavior of the system in an attempt to measure δ_v. The samples were taken at 10 second intervals for the first five minutes and at 1 minute intervals for the next two hours. From Figure 6, we observe that the drift remains nearly constant after a short period of time for our experimental setup and for the idle Ubuntu and Vista systems, the drift stabilizes within a few minutes. The drift for a system running Mozilla Firefox was found to be nearly constant within 10 minutes. Running *tar* and *gzip* for compressing a large folder or *dd* to dump the contents of memory into a file resulted in most of the memory being changed within a minute due to second-order taint. To summarize, our tests demonstrated that there is no taint introduced in the extended memory by using Forenscope and that Forenscope can be used for forensic analysis where taint needs to be minimized.

Effectiveness against anti-forensics tools: Although forensics techniques can collect significant amounts of information, investigators must be careful to ensure the veracity and fidelity of the evidence collected because anti-forensic techniques can hide or intentionally obfuscate information gathered. In particular, rootkits can be used by hackers to hide the presence of malicious software such as bots running in the system. Malware tools such as the FU rootkit [16] directly manipulate kernel objects and corrupt process lists in ways that many tools cannot detect.

Table 5: Sizes of Forenscope and modules

Component	Lines of Code	Compiled Size (bytes)
Forenscope (C)	1690	15,420
Forenscope (Assembly)	171	327
Forenscope (Hardware)	280	1,441
Neutralizer & Forenshell	34	8,573
Other Modules	861	22,457
Total	3,036	48,218

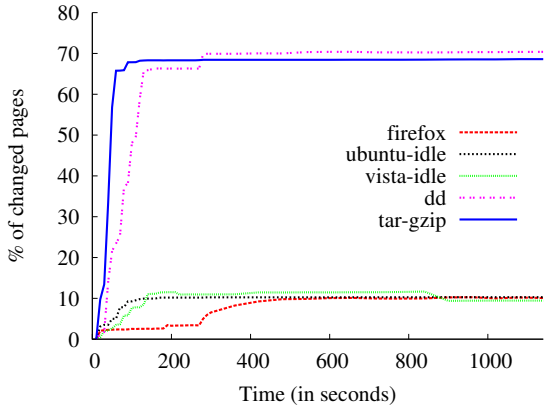

Figure 6: Comparison of Memory Blurriness

Malware researchers have also demonstrated techniques to evade traditional memory analysis through the use of low-level rootkits [28] which cloak themselves by deceiving OS-based memory acquisition channels on Linux and Windows. Hardware [12] and software [20] virtualization-based rootkits may be tricky to detect or remove by the legitimate operating system or application software because they operate one layer below standard anti-malware facilities. We describe and evaluate how Forenscope reacts to several publicly available rootkits. The set of rootkits was chosen to cover a gamut of representative threats, but the list is not meant to be exhaustive due to space constraints.

DR: The DR rootkit uses processor-level hardware debug facilities to intercept system calls rather than modifying the actual system call table itself. DR reprograms a hardware breakpoint which is reached every time a system call is made [15]. The breakpoint then intercepts the call and runs its own handler before passing control to the legitimate system call handler. Since Forenscope does not restore the state of debug registers, DR is effectively neutralized across the reboot, and as a result, hidden processes are revealed. Informant detects DR in several ways: DR is present in the module list, DR symbols are exported to the kernel and DR debug strings are present in memory. If an attacker modifies DR to make it more stealthy by removing these indicators, we contend that it is still hard to deceive Forenscope, since the debug registers are cleared as part of the reboot process. Although Forenscope doesn't restore the contents of the debug registers faithfully, this doesn't pose a problem for most normal applications because only debuggers typically use this functionality.

Phalanx B6: Phalanx hijacks the system call table by directly writing to memory via the */dev/mem* memory device. It works by scanning the internal symbol table of the kernel and redirecting control flow to its own internal functions. Informant detects Phalanx while checking the system call table and common kernel pointers. Neutralizer restores the correct pointers to inoculate Phalanx.

Adore: Adore [8] is a classic rootkit which hijacks kernel pointers to deceive tools such as *ps* and *netstat*. It works by overwriting pointers in the */proc* filesystem to redirect control flow to its own functions rather than modifying the syscall table directly. Informant detects that the pointers used by Adore do not belong to the original read-only program code segment of the kernel and Neutralizer restores the correct pointers. Restoration of the original pointers is simple and safe because the overwritten VFS function operations tables point to static functions such as *proc_readdir*, while Adore has custom handlers located in untrusted writable kernel module address space.

Mood-NT: Mood-NT is a versatile multi-mode rootkit that can hook the system call table, use debug registers and modify kernel pointers. Because of its versatility, the attacker can customize it for different purposes. Like the rootkits described previously, Forenscope detects Mood-NT in various modes. Our experiments indicate that Mood-NT hooks 44 system calls and Forenscope detects all of these alterations. Furthermore, each hook points out of the kernel's read-only program code address space and into the untrusted memory area occupied by the rootkit.

Size: Forenscope is written in a mixture of C and x86 assembly code. Table 5 shows that Forenscope is a very small program. It consumes less than 48 KB in code and 125 KB in running memory footprint. The lines of code reported in the table are from the output of the *sloccount* [29] program. We break down the size of each component into core C and assembly code, hardware-specific restoration code and module code. To minimize its size, Forenscope reuses existing kernel code to reinitialize the disk and network; the size of this kernel code is device-specific and therefore excluded from the table, since these components are not part of Forenscope. The small compiled size of Forenscope and its modules implies that a minimal amount of host memory is overwritten when Forenscope is loaded onto the system. Furthermore, the diminutive size of the code base makes it more suitable for auditing and verification.

5. DISCUSSION

While evaluating Forenscope, we observed different behavior of rootkits on virtual machines and physical hardware. Our observations confirm the results of Garfinkel et al [17] that virtual machines cannot emulate intricate hardware nuances faithfully and as a result some malware fails to activate on a virtual machine. For example, malware such as the Storm worm and Conficker [30] intentionally avoid activation when they sense the presence of virtualization to thwart the analysis process. Hence analyzing a system for rootkits using a virtual machine may not only cause some rootkits to slip under the radar but also alert them to detection attempts. Since Forenscope continues to run the system without exposing any of the issues raised by running virtualization systems, we argue that the system is unlikely to tip off an attacker to the presence of forensic software. Legally, the jury is still out on the use of live forensic tools because of the issues of taint and blurriness. While some recent cases [2] suggest that courts are starting to recognize the value of the contents of volatile memory, the validity of the evidence is still being contested. A recent manual on collecting evidence in criminal investigations released by

[8] http://stealth.openwall.net

Table 6: Effectiveness against rootkit threats

Rootkit	Description	Sanitization action
DR	Uses debug registers to hook system calls	Rebooting clears debug registers
Phalanx b6	Uses /dev/kmem to hook syscalls	Restore clean syscall table
Mood-NT	Multi-module RK using /dev/kmem/	Clear debug regs, restore pointers
Adore	Kernel module hooks /proc VFS layer	Restore original VFS pointers

the Department of Justice [6], instructs that *no limitations should be placed on the forensic techniques that may be used to search* and also states that use of forensic software, no matter how "sophisticated," does not affect constitutional requirements. Although we do not make strict claims of legal validity in the courts, we are encouraged by the above guidelines to collect as much volatile information as possible. We objectively compare our tool against the state of the art and find that it does collect more forms of evidence with better fidelity than existing tools.

Countermeasures: Although Forenscope provides deep forensic analysis of a system in a wide variety of scenarios, there are countermeasures that attackers and criminals can use to counter the use of Forenscope. From an incident response perspective, we assume that the machine is controlled by the owner and that the attacker does not have physical access to it. This means that only software-based anti-forensic techniques are feasible, although some of these techniques may involve changing hardware settings through software. Most of the hardware and software state involved in these anti-forensic techniques are cleared upon reboot or rendered harmless in Forenscope's clean environment. In investigation, the adversary may elect to use a BIOS password, employ a secure bootloader, disable booting from external devices or change BIOS settings to clear memory at boot time. These mitigation techniques may work, but if the investigator is sophisticated enough, he can try techniques suggested by Halderman et al [19] to cool the memory chips and relocate them to another machine which is configured to preserve the contents of DRAM at boot time. One other avenue for working around a password-protected BIOS is to engage the bootloader itself. We found that some bootloaders such as GRUB allow booting to external devices even if the functionality is disabled in the BIOS. The only mitigation against this channel is use password protection on GRUB itself, which we believe is not frequently used.

Limitations: The only safe harbor for malware to evade Forenscope is in conventional memory itself because the act of rebooting pollutes the contents of the lower 640 KB of memory considerably thus potentially erasing evidence. However, we contend that although this technique is possible, it is highly unlikely for three reasons: first, for such malware to persist and alter the control flow, the kernel must map in this memory area in the virtual address space. This requires a change in the system page tables which is easily detectable by Forenscope since most modern operating systems do not map the conventional memory space into their virtual memory space. Secondly, such malware would have to inject a payload into conventional memory and if the payload is corrupted by the reboot process, the system will crash. Finally, such malware won't survive computer hibernation because conventional memory is not saved in the process. Even if Forenscope is unable to restore the system due to extenuating circumstances, we still have an intact memory dump and disk image to analyze. Although Forenscope has been designed with investigation in mind, we have not designed it to be completely transparent. For instance, malware might detect the presence of Forenscope by checking BitBlocker write latencies or scanning conventional memory.

6. RELATED WORK

Forenscope uses many technologies to achieve a high fidelity forensic analysis environment through introspection, data structure analysis and integrity checking. Many of the introspective techniques used by Forenscope were inspired by similar functionality in debuggers and simulators. VMware's VMsafe protects guest virtual machines from malware by using introspection. A virtual machine infrastructure running VMsafe has a security monitor which periodically checks key structures in the guest operating system for alteration or corruption. Projects such as Xenaccess [22] take the idea further and provide a way to list running processes, open files and other items of interest from a running virtual machine in a Xen environment. Although Xenaccess and Forenscope provide similar features, Xenaccess depends on the Xen VMM, but the investigator cannot rely on its presence or integrity. On some older critical infrastructure machines, legacy software requirements make it impractical to change the software configuration. Forenscope does not have such requirements. Forenscope's techniques to recover operating system state from structures such as the process list have been explored in the context of analyzing memory dumps using data structure organization derived from reverse-engineered sources [14, 27]. Attestation shows that a machine is running with an approved software and hardware configuration by performing an integrity check. Forenscope builds upon work from the VM introspection community to allow forensic analysis of machines that are not prepared a priori for such introspection. It provides a transparent analysis platform that does not alter the host environment and Forenscope supports services such as BitBlocker that allow an investigator to explore a machine without inducing taint.

The techniques used by Forenscope for recovering running systems are well grounded in the systems community and have been studied previously in different scenarios. The original Intel 286 design allowed entry into `protected mode` from `real mode`, but omitted a mechanism to switch back. Microsoft and IBM used an elegant hack involving memory remanence to force re-entry into real mode by causing a reboot to service BIOS calls. This technique was described by Bill Gates as "turning the car off and on again at 60 mph" [24]. Some telecommunications operating systems such as Chorus [25] are designed for quick recovery after a watchdog reset and simply recover existing data from the running operating system rather than starting afresh. David [13] showed that it is possible to recover from resets triggered by the watchdog timer on cell phones. BootJacker [11] showed that it is possible for attackers to recover and compromise a running operating system by using a carefully crafted forced reboot. Forenscope applies these techniques in the context of forensic analysis and our work presents the merits and limitations of using such techniques to build a forensic tool.

Devices such as the Trusted Platform Module and Intel trusted execution technology (TXT) provide boot time and run-time attestation respectively. Although TPM may be available for some machines, the protection afforded by a TPM may not be adequate for machines which are meant to run continuously for months. These machines perform an integrity check when they boot up, but their lengthy uptime results in a long time of check to time of use (TOCTTOU) that extends the duration for breaches to remain undetected. Hardware solutions such as Copilot [23] are available to check system integrity. In contrast, Forenscope performs an integrity assessment at the time of use; which allows the investigator to collect evidence with better fidelity.

7. CONCLUDING REMARKS

Forenscope explores live forensic techniques and the issues of evidence preservation, non-intrusiveness and fidelity that concern such approaches. Measured against existing tools, our experiments show that Forenscope can achieve better compliance within the guidelines prescribed by the community. Forenscope shows that volatile state can be preserved and the techniques embodied in Forenscope are broadly applicable. We encourage further development of tools based on our high-fidelity analysis framework and believe that it can enable the advancement of analysis tools such as KOP [9]. Extensive evaluation of our techniques has shown that they are safe, practical and effective by minimally tainting the system, while causing no disruption to critical systems. We believe that these techniques can be used in cases where traditional tools are unable to meet the needs of modern investigations. To continue the development of this tool, we plan to work closely with partners to better evaluate use of this tool in real-world scenarios such as incident response in a variety of contexts.

Acknowledgements We would like to thank the anonymous reviewers, Winston Wan, Mirko Montanari and Kevin Larson for their valuable feedback. This research was supported by grants from DOE DE-OE0000097 under TCIPG (tcip.iti.illinois.edu) and a Siebel Fellowship. The opinions expressed in this paper are those of the authors alone.

8. REFERENCES

[1] SANS Top 7 New IR/Forensic Trends In 2008. `http://computer-forensics.sans.org/community/top7_forensic_trends.php`.

[2] Columbia Pictures Indus. v. Bunnell, U.S. Dist. LEXIS 46364. C.D. Cal. `http://www.eff.org/cases/columbia-pictures-industries-v-bunnell`, 2007.

[3] *Prosecuting Computer Crimes*, pages 141–142. US Department of Justice, 2007.

[4] Electronic Crime Scene Investigation: A Guide for First Responders. pages 25–27, 2008.

[5] Ramdisks - Now We are Talking Hyperspace! `http://www.linux-mag.com/cache/7388/1.html`, 2009.

[6] *Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations*, pages 79,89. Computer Crime and Intellectual Property Section Criminal Division, 2009.

[7] K. Amari. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory, 2009.

[8] D. Brezinski and T. Killalea. Guidelines for Evidence Collection and Archiving. RFC 3227 (Best Current Practice), Feb. 2002.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565, New York, NY, USA, 2009. ACM.

[10] C. C. Center. How the FBI Investigates Computer Crime. `http://www.cert.org/tech_tips/FBI_investigates_crime.html`, 2004.

[11] E. Chan, J. Carlyle, F. David, R. Farivar, and R. Campbell. BootJacker: Compromising Computers using Forced Restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 555–564. ACM New York, NY, USA, 2008.

[12] D. Dai Zovi. Hardware Virtualization Rootkits. *BlackHat Briefings USA, August*, 2006.

[13] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[14] B. Dolan-Gavitt. The VAD tree: A Process-eye View of Physical Memory. *Digital Investigation*, 4:62–64, 2007.

[15] Edge, Jake. DR rootkit released under the GPL. `http://lwn.net/Articles/297775/`.

[16] Fuzen Op. The FU rootkit. `http://www.rootkit.com/project.php?id=12`.

[17] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[18] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–90, July 1996.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, and J. A. Calandrino. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[20] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[21] W. Link and H. May. Eigenshaften von MOS-Ein-Transistorspeicherzellen bei tieften Temperaturen. In *Archiv fur Elektronik und Ubertragungstechnik*, pages 33–229–235, June 1979.

[22] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of 23rd Annual Computer Security Applications Conference*, pages 385–397, 2007.

[23] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot-A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[24] J. Pournelle. OS | 2: What is is, What is isn't – and some of the Alternatives. *Infoworld*, 1988.

[25] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1991.

[26] A. Savoldi and P. Gubian. Blurriness in Live Forensics: An Introduction. In *Proceedings of Advances in Information Security and Its Application: Third International Conference, Seoul, Korea*, page 119. Springer, 2009.

[27] A. Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. The Proceedings of the 6th Annual Digital Forensics Research Workshop, 2006.

[28] S. Sparks and J. Butler. Raising The Bar for Windows Rootkit Detection. *Phrack*, 11(63), 2005.

[29] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount`.

[30] B. Zdrnja. More tricks from Conficker and VM detection. `http://isc.sans.org/diary.html?storyid=5842`, 2009.