2009

Towards virtual machine integrity using introspection

Sammy Lin

Towards Virtual Machine Integrity using Introspection

By

Sammy Lin

Thesis submitted in partial fulfillment of the requirements
for the degree of
Master of Science in
Computer Security and Information Assurance

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

April 4, 2009

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Computer Security and Information Assurance

Thesis Approval Form

Student Name: Sammy Lin

Thesis Title: Towards Virtual Machine Integrity using Introspection

Thesis Committee

Name	Signature	Date
Charles Border Chair		
Bo Yuan Committee Member		
Luther Troell Committee Member		

Abstract

The Integrity Measurements Architecture (IMA) [5] provides attestation and integrity for Linux hosts. But what if an administrator wants to provide IMA functionality to an older (non-IMA capable) or a non Linux-based OS? If the system is deployed on top of a hypervisor, IMA functionality can be provided at the hypervisor level. This paper applies Virtual Machine Introspection (VMI) to provide IMA functionality to virtualized guest OSes. We implement a proof of concept library (using a shallow shadow filesystem) and integrate it with the Kernel-based Virtual Machine (KVM) hypervisor. The modifications provide the Linux host OS the ability to see when and what files are being accessed by the guest OS. This paper outlines the approach to its design, concept of execution, and describes the challenges encountered. The library is tested with a sample bash script created in a monitored partition; a hash of the file is printed before the file is loaded into memory.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Today, more systems owners are realizing their hardware is being under-utilized, and virtualization is becoming a common solution. Virtualization allows multiple server instances to exist on the same hardware, reducing Size, Weight, and Power (SWAP), while still providing isolation. In a virtualized environment, a new layer is introduced between the hardware and the server software. This virtualization layer provides an environment to monitor the activity of virtual machines. This monitoring code lives outside the guest OS, making it less susceptible to attacks. While being isolated from the guest OS provides a security advantage, a high level view of its activity is lost. The challenge lies in rebuilding this view and being able to identify guest OS activity (e.g., disk activity, process loading). Providing this of view in to the VM is called Virtual Machine Introspection. VM Introspection can be applied to provide various security benefits to VMs [1] [2] [3]; one of them being integrity measurements.

The virtualization layer handles requests from VMs for data on a hard disk drive. The guest OS makes requests for data to this layer, and uses sector numbers to specify which unit of data it's requesting. This research builds upon the fundamental theory of building a table of logical block numbers, and using that table to map blocks to logical files. Being able to do this provides the virtualization layer a view into what files are being accessed by the guest OS. In order to realize this theory, the filesystem to be monitored needs to be parsed and its meta-data stored. More specifically, for each file in the filesystem, its logical blocks need to be recorded, essentially building a shallow shadow filesystem.

The goal of this research is to be able to measure files (of the virtual disk) before they're loaded by their respective VM. A measurement is a SHA-1 hash of the file contents. A shallow shadow filesystem is used, in contrast to a deep shadow filesystem, which is a bit for bit copy. Our approach uses only the meta-data of a target filesystem, and doesn't need to clone the entire filesystem. A proof of concept is developed to illustrate the feasibility and challenges associated with this application of VMI. For our proof of concept the Kernel-based Virtual Machine (KVM) is used for the hypervisor, Gentoo Linux as the guest OS, and the Second Extended Filesystem (EXT2) for the filesystem.

The scope of this research is limited to the notification of files being accessed, and hashing the contents. File writes will not be a part of this research, but will be discussed in Challenges and Future Work (7).

In the next sections (2, 3, and 4), we give an overview of IMA, KVM, and the EXT2 filesystem. We outline the proof of concept and explanation of the environment in section 5. We realize that the work in this paper is limited in scope, and is only scratching the surface of integrity in Virtual Machines; we outline the challenges that were encountered and possible future work in section 7.

1.1 Related Work

Currently, COTS hypervisors already provide VM Introspection utilities. VMware offers a proprietary technology for Virtual Infrastructure called VMSafe [11]. This technology allows 3rd party vendors to develop security tools that take advantage of VM Introspection. VMSafe allows a security application to view the contents of Memory pages, CPU state, network packets, and storage, but didn't appear to have the ability to examine a file before it is loaded.

Payne, Carbone, and Lee implemented the XenAccess [7] monitoring library for VMs using the Xen hypervisor. Their research allowed for introspection of virtual memory and disk. XenAccess is similar to the work presented in this paper, because the inference engine is dependant on knowledge of the filesystem in use [7]. So far, knowledge has been included in the inference engine to be able to determine only file/directory creation/removal operations under the ext2 filesystem, although knowledge about other filesystems can be incorporated [7]. We have taken their research one step further, by allowing the hypervisor to know which files are loaded.

Kourai and Chiba implement a solution called HyperSpector [9], which uses a full shadow filesystem. Instead of providing a shallow copy of the monitored filesystem, HyperSpector creates a full clone, and has a mediator that handles I/O requests from the shadow-filesystem to the real filesystem.

2 IMA Overview

The Integrity Measurements Architecture (IMA) [6] motivates the research presented in this paper. IMA is the implementation of a secure integrity measurement system for Linux. Integrity and attestation play an important role in high assurance applications [10] [6]. IMA provides an OS with the ability to passively take measurements on all files loaded in to memory. IMA follows the operating principle of, "measuring software before loading it". A system following this principle would record the fact that software has been loaded, even if it is malicious. Malicious software cannot change this fact because the measurement is taken prior to being loaded, and IMA maintains the measurements list in the kernel.

In TPM-based Attestation, after a value is added to the measurement list, the list is hashed and the value is extended in Platform Configuration Register (PCR - normally 160 bits) #10 of the Trusted Platform Module (TPM) hardware chip [11]. The Trusted Platform Module is a secure crypto-processor that store cryptographic keys, and other valuable information. The PCR always contains an aggregate of the current measurement list. PCR values of the TPM cannot be overwritten directly; they can only be extended. The extend operation of the PCR is handled by taking the new value and hashing it with the previous, as described in Figure 2-1.

```
PCR := SHA-1(OldVal + measurement)
```

Figure 2-1: PCR Extend Operation

Using the measurements list, and the aggregate value (hash of list) in the PCR, a verifier can recreate (or verify) the software that was loaded on the target machine. A verifying host would perform the following to validate an IMA enabled host:

- Gather measurement list and value held in PCR #10
- For each measurement in the list, generate a hash of the measurement along with the previous value in the PCR.

After the entire measurement list has been processed, the value computed is compared to the value gathered from PCR #10. The verifier is essentially recreating the current PCR value and so it can make a decision on the validity of software running on the host.

The table below is a sample of the entries of an integrity measurements list.

Table 2-1: Sample Measurement List [17]

#007	CFBC7EC3302145AB78A307C0D41DBB9A4251377B	mmap-file	libnss_files-2.3.2.so	Library
#008	805572455CF5BF50A7EE42E3CC6B0EDA65AF17A4	mmap-file	initlog	Executable
#009	C95CBC5625719649103E0D1C3595967474842F7B	mmap-file	hostname	Executable
#010	0CAA342424F420FF29B7FB2FCF278F973600681B	mmap-file	mount	Executable

In an IMA enabled host, the TPM would be the hardware root of trust, and a verifier does not have to rely on the trustworthiness of the software environment. The only caveat of IMA is it's only available for capable Linux kernels. Systems using older Linux kernels would need a back port of IMA for that specific kernel version. This doesn't take into account systems that aren't Linux based. For this research, we pull IMA functionality out of the VM and use the virtualization layer as the root of trust, since it is the mechanism that takes the measurements. Pulling the IMA functionality out of the guest OS removes the OS type restriction as well.

3 KVM Background

KVM, is a Linux virtualization infrastructure, which provides a native virtualization environment but requires modifications to the host OS (the KVM kernel module), in contrast to paravirtualization which requires no modification to either guest or host (e.g., VMware Workstation). KVM uses a modified version of the QEMU emulation software to provide address space, simulated I/O, and mapping display back to the host. The most relevant portion of QEMU to this research is the I/O code. This I/O code is where we inject lookup code to our shadow filesystem. It is also where we will call our measurement code.

KVM parent company Qumranet was recently (Sept. 2008) acquired by RedHat, and the kernel component of KVM is included in mainline Linux, as of 2.6.20.

4 EXT2 Background

For this research, EXT2 is selected as the filesystem. EXT2 implements a common set of UNIX filesystem concepts, the most relevant concept to this research being the inode.

The inode represents files and directories and contains all meta-data related to a file including:

- Type
- Permissions
- Timestamps (access, create, modified)
- Size
- Pointers to data

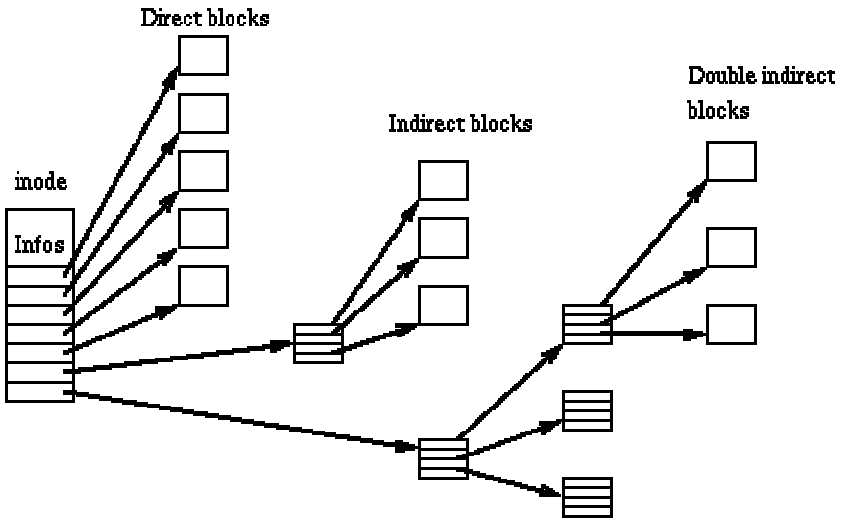

Figure 4-1: Blocks of Inode [23]

Figure 4-1 illustrates the structure of the inode. `Infos` represent the meta-data described above. The rest of the inode contain 32-bit pointers to the data blocks. Each inode has room for 15 addresses (represented by the `EXT2_N_BLOCKS` macro), but only elements 0 to 11 store addresses of actual data blocks. Block 12 contains the address to another table of block addresses called indirect blocks; block 13 contains an address to a table of addresses to tables of addresses of data blocks called double indirect blocks; block 14 holds triple indirect blocks.

In order to read the inodes of an EXT2 filesystem, a library called `libext2fs` is required. Many of the Linux filesystem tools (e.g., `mke2fs`, `e2fsck`) use this library to carry out their duties. This library contains functions to read the superblock, inodes, data blocks, and other data types of the filesystem. For this research, we are interested in

iterating through each inode and recording the block numbers that it contains. The `libext2fs` library is contained in the Linux package `e2fsprogs`.

5 Proof of Concept

This research develops a proof of concept to illustrate the feasibility of being able to identify a file is being loaded into memory by knowing its blocks. In this section we describe our hardware and software environment, how the virtual disk is constructed, the design of the lookup table, and how the library works.

5.1 Environment

For the proof of concept, the following hardware and software system configurations is utilized:

Table 5-1: Hardware System Configuration

Hardware Item	**Configuration**
Northbridge Chipset	Intel Q35
CPU	Intel E6750
RAM	3GB PC5300
Hard Disk	2x250GB

The hardware was specifically chosen because the E6750 CPU offers Intel® Virtualization Technology (VT). AMD also has hardware virtualization technology called AMD-V™. In order to take advantage of this CPU feature, the Q35 was required. Rather than build a custom machine, this machine was procured from Lenovo using readily available configurations. Intel® VT is needed because KVM requires this feature to provide native virtualization.

Table 5-2 describes the software configuration for development of the proof of concept. Open source software is used when available, and trial licenses were obtained as needed. 64 bit Redhat Enterprise Linux Server 5 is used as the host OS, and Gentoo Linux (2008.0 profile) is used as the guest. The reason for the difference in OSes is because we didn't want an installer to partition our virtual disk for us; also, we didn't want to use Logical Volume Manager (LVM). `e2fsprogs` provides `libext2fs` and OpenSSL provides the cryptographic library.

Table 5-2: Software System Configuration

Software Item	Configuration
Host OS	Redhat Enterprise Linux Server 5 x86_64
Hypervisor	Kernel-based Virtual Machine-77
Guest OS	Gentoo Linux 2008.0
Guest OS Filesystem	ext2/3
Guest OS Virtual disk format	raw
Filesystem library	e2fsprogs-1.39
Cryptographic library	openssl-0.9.8j

5.2 The Virtual Disk

Like host OSes, VMs need a disk to boot from and store its files. VMs usually use a large file to represent the virtual disk. This virtual disk will represent the hard disk in which KVM can boot from and store guest OS files. KVM comes with a tool called `qemu-img`, which creates the virtual disk in various formats. For portability and simplicity reasons, we've chosen to use the default raw image format for our virtual disk. The raw format is a plain binary image of the hard disk, and supports copying directly to and from the host OS.

To create the virtual disk the following command is used:

```
#qemu-img create vdisk.img 10GB
```

Figure 5-1: Image Creation Command

This command allocates a 10GB virtual disk to a file called `vdisk.img` using the raw format. The raw format is used by default, which is why no command line option is used.

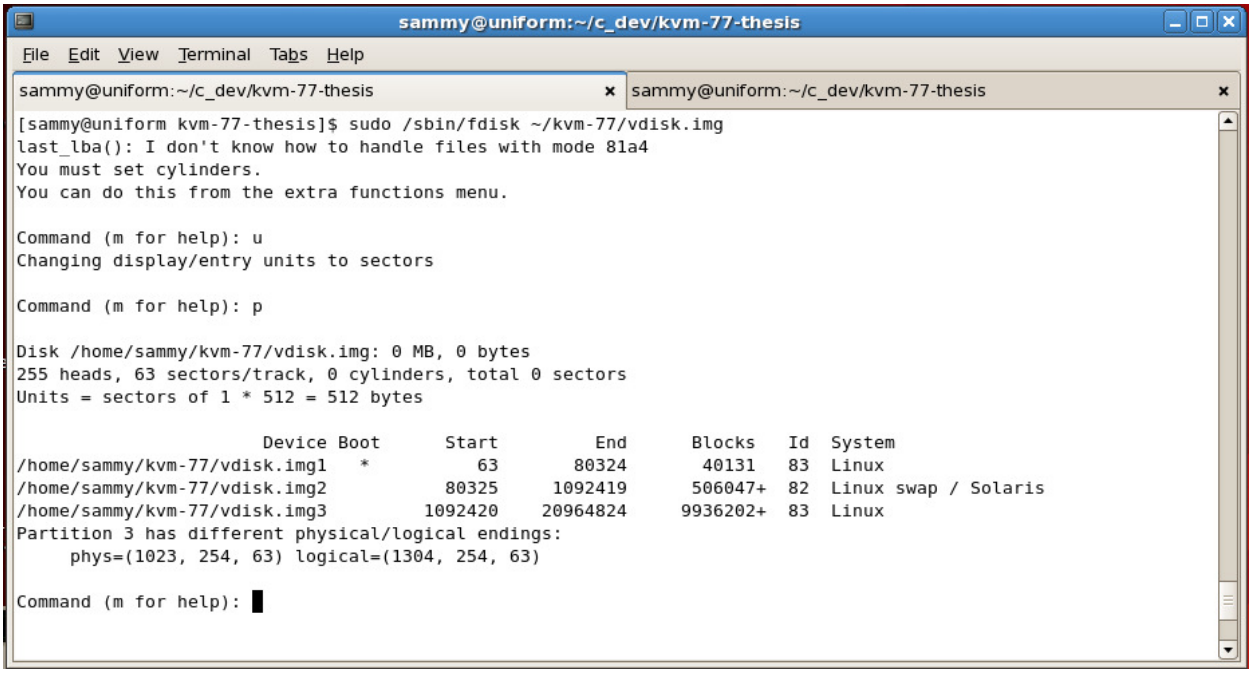

Figure 5-2: `fdisk` Output

Once the virtual disk is created, partitions are created and formatted. Figure 5-2 illustrates the output from `fdisk` for the virtual disk. A traditional Linux partition scheme is chosen for simplicity, which consists of three partitions: boot, swap, and root ('/'). The boot partition uses an ext2 filesystem, while the root partition utilizes an ext3 partition with journaling.

In order to handle this virtual image like a physical disk, a couple of technologies will need to be used. The mount command is used to mount a filesystem, but since this is not a physical disk, a couple of options need to be used. If the virtual disk were to be mounted on a Linux system, the following command would be used:

```
#mount -o loop,offset=32256 vdisk.img /mnt/image
```

Figure 5-3: Mount Boot Partition Command

This command makes an internal call to `losetup` to create the loop device and use an offset into the disk image. The offset determines which partition is to be mounted. In order to come up with the number for the offset, a couple of parameters need to be gathered: (block size, partition start position). Looking at the partition scheme in Figure 5-2, and using the `u` command in `fdisk` to switch the units to sectors, we can see that the

boot partition starts at sector 63. Each sector is 512 bytes, which means the boot partition begin at 32256 bytes from the start of the file. The formula for the calculation is as follows:

$$(63 \times 512) = 32556$$

If the root partition were to be mounted, the following command would be used:

```
#mount -o loop,offset=41126400 vdisk.img /mnt/image
```

Figure 5-4: Mount Root Partition Command

For this proof of concept, we're limiting the shadow filesystem to the boot partition since it contains fewer files.

Figure 5-5 illustrates the disk and shows the physical layout of the disk. The test file `ps.sh` resides in the boot partition.

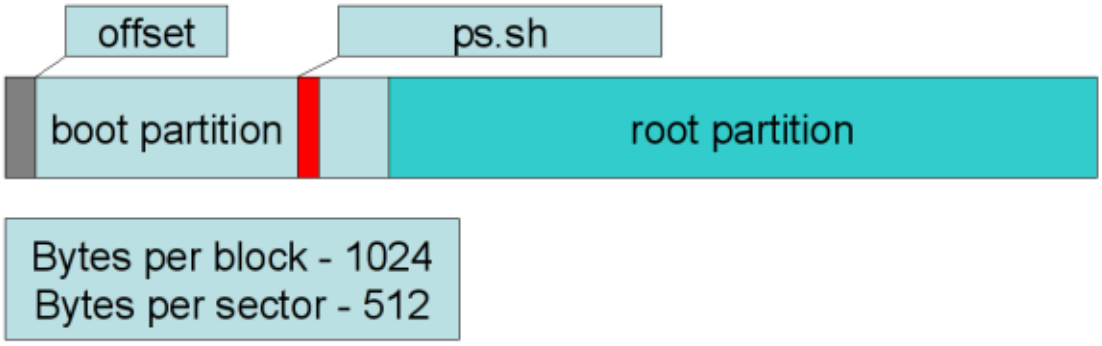

Figure 5-5: Disk Layout

5.3 Design

The following section describes the detailed design for the library that is developed to support the shadow filesystem and the shadow filesystem implementation itself.

5.3.1 The Shadow Filesystem

The shadow filesystems main component is the lookup table. The lookup table is composed of 3 core structures: `s_inode`, `s_block`, and `s_shadow_fs`.

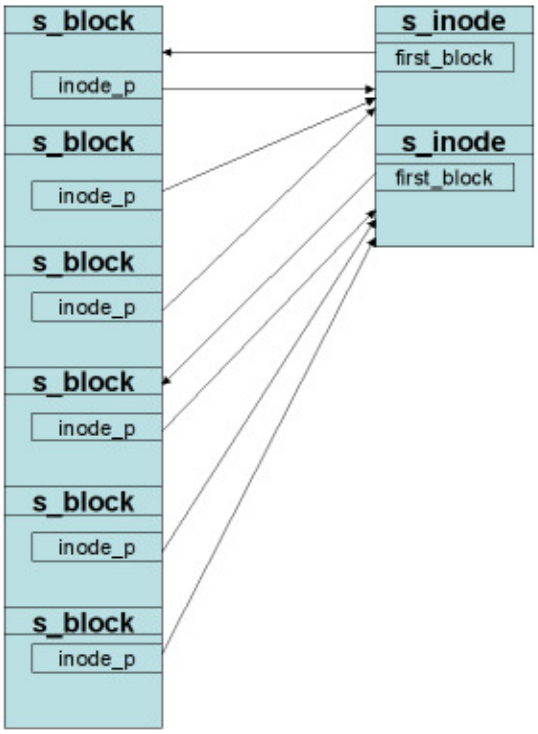

Figure 5-6: Lookup Table

Figure 5-6 illustrates the look up table for our shadow filesystem. The array of blocks contains pointers to the inode in which it belongs to, allowing quick reverse lookups for any block. Each inode in the array of inodes contain a pointer to the first block of the inode as described later (`s_inode`).

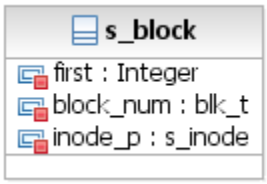

Figure 5-7: `s_block` Structure

The `s_block` structure encapsulates the meta-data for a block of data. The first attribute of the structure is a Boolean to determine if the block is the first of a group of blocks of a file. KVM receives requests for blocks independently, making it necessary to add logic to determine which block represents a file. Without this logic, each block request would look like a file request. For example, given a file `fileA` (made up of blocks {1, 2, 3}) that is loaded into memory, KVM would see requests for blocks 1, 2,

and 3. Without the first Boolean, the shadow filesystem would interpret that as three separate attempts to access `fileA`. Using the first Boolean, we would only take into account when the first block is loaded, making the assumption that the VM is trying to load `fileA`, because files normally load bytes in sequence. Only requests for blocks that have the `first` attribute set to `1` are processed, while others are discarded. `block_num` is a number of the type `blk_t` (from `libext2fs`) and represents the block number. The `inode_p` attribute is a pointer to the inode this block belongs to.

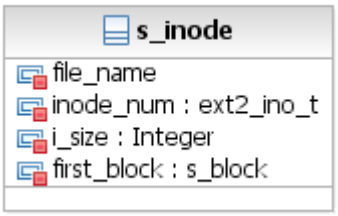

Figure 5-8: `s_inode` Structure

The `s_inode` structure represents the meta-data for an inode, currently the `file_name` attribute serves as a placeholder for a meaningful file name. The EXT2 filesystem represents files as inodes and inodes can have multiple names associated with it. Associating a filename would give more meaning than an inode number. `inode_num` represents the inode number, which is most relevant, but not as useful as a filename. `i_size` represents the size of the file, which is used for hashing its contents. The `first_block` attribute is a pointer to the first block for this particular inode.

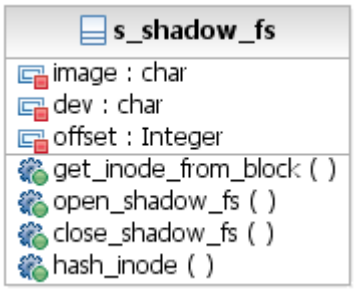

Figure 5-9: `s_shadow_fs` Structure

The `s_shadow_fs` structure encapsulates our shadow filesystem. It contains information relevant to the filesystem it is shadowing. `image` is a string representation

of the path for the hard disk image, `dev` represents the loop device the virtual disk image is attached to, and `offset` represents the offset into the virtual disk. It also includes pointers to functions which setup/close as well as provides lookup information.

The `open_shadow_fs` function contains functionality to create the loop device, read the filesystem information, and build the lookup table. The `get_inode_from_block` function gets the inode given a block number. The `hash_inode` function hashes the contents of an inode and prints the SHA-1 hash out to console. `close_shadow_fs` performs cleanup and removes the loop device.

5.4 Concept of Execution

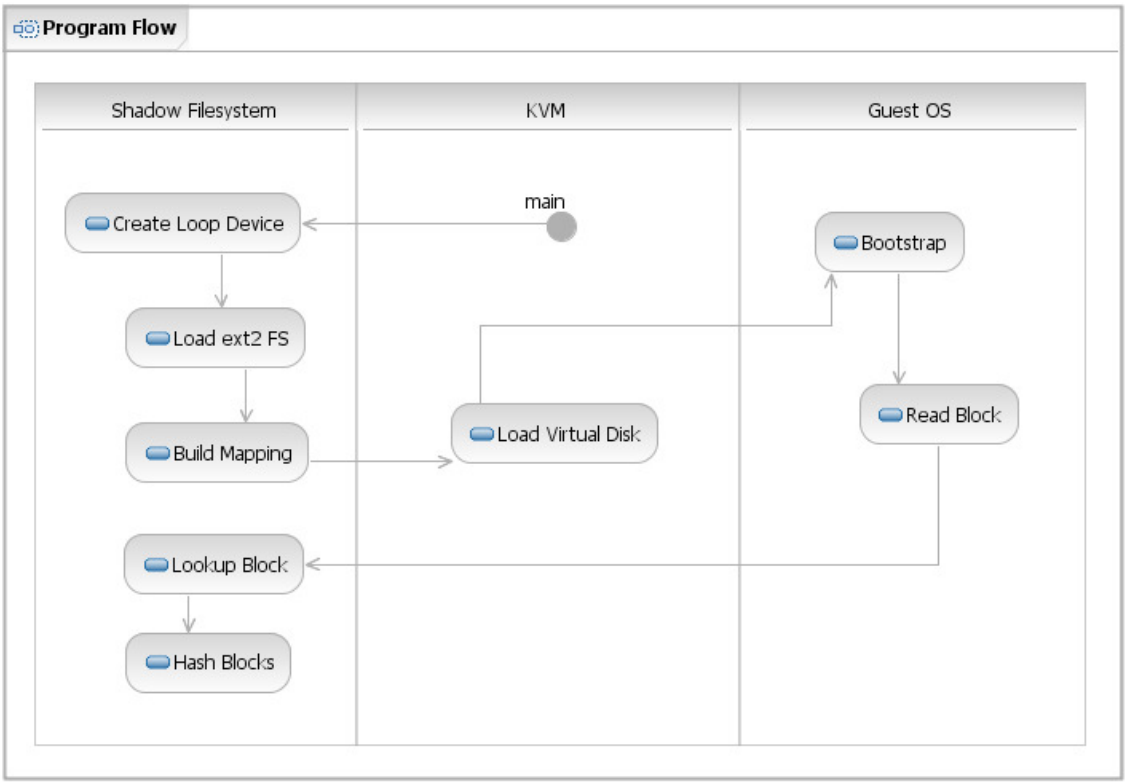

Figure 5-10: Software Flow Diagram

Figure 5-10 illustrates the flow of execution for our proof of concept. In order to take advantage of Intel® VT and the hardware virtualization technology, the `kvm` and `kvm-intel` kernel modules need to be loaded into the kernel.

```
#insmod kernel/x86/kvm.ko
#insmod kernel/x86/kvm-intel.ko
```

Figure 5-11: Kernel Module Install Commands

We launch the hypervisor by executing the following command, specifying our virtual disk using the `hda` option and allocate 384MB of ram for the guest OS.

```
#qemu-system-x86_64 -hda vdisk.img -m 384
```

Figure 5-12: Starting KVM Command

Once KVM has been launched (initial state in flow diagram), control is passed to the shadow filesystem to read and process the contents of the original filesystem. It creates a loop device so it can read the file as a block device. The filesystem is loaded and a mapping is built (detailed in section 5.5). Control is passed back to KVM to load the virtual disk and let the guest OS use it to boot. When the guest OS is operational, the shadow filesystem is ready to service requests for lookup. When the guest OS makes a request for a block, control is passed to the shadow filesystem to lookup the blocks and determines which inode is being accessed. This information is collected, the contents of the inode is hashed and reported back to the host OS via console. Figure 5-13 illustrates what is happening inside the guest OS, while Figure 5-14 illustrates the host OS. The guest OS mounts the boot partition (the partition we're monitoring), and executes the file (`/boot/bin/ps`). This creates the output on Figure 5-13. The guest OS continues to print information pertaining to the file to verify the proper inode has been hashed. A directory listing is performed to include inode numbers and the inode for `/boot/bin/ps` is `8037`. Also, inside the guest OS, a hash is generated for the file and produces the same output as our shadow filesystem, validating our proof of concept.

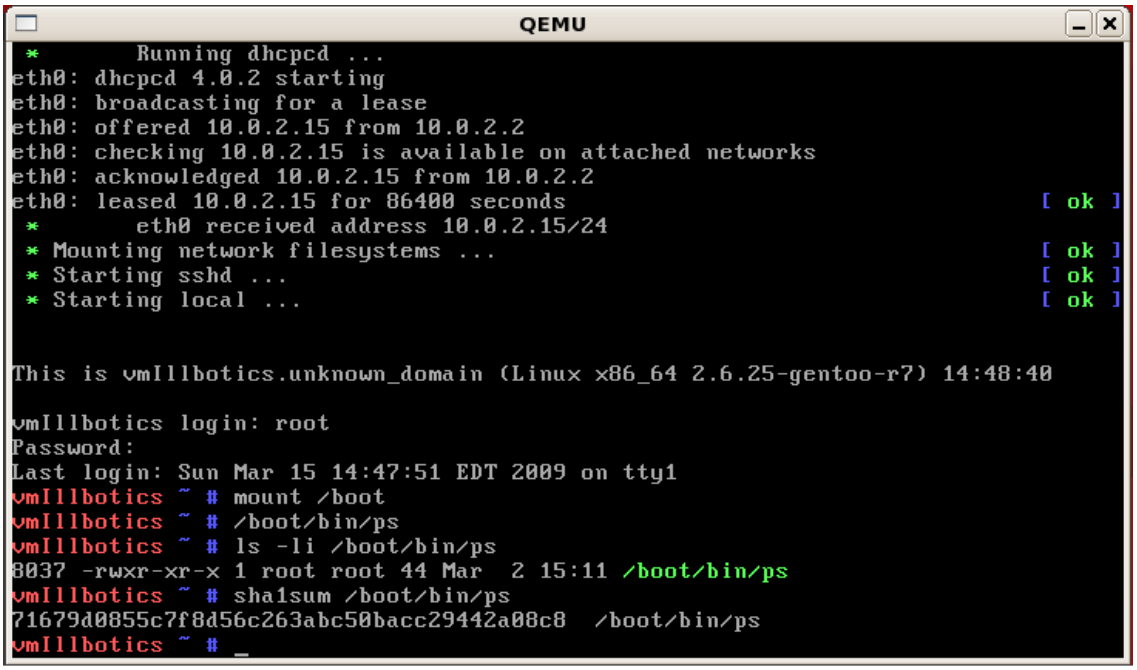

Figure 5-13: Guest OS Output

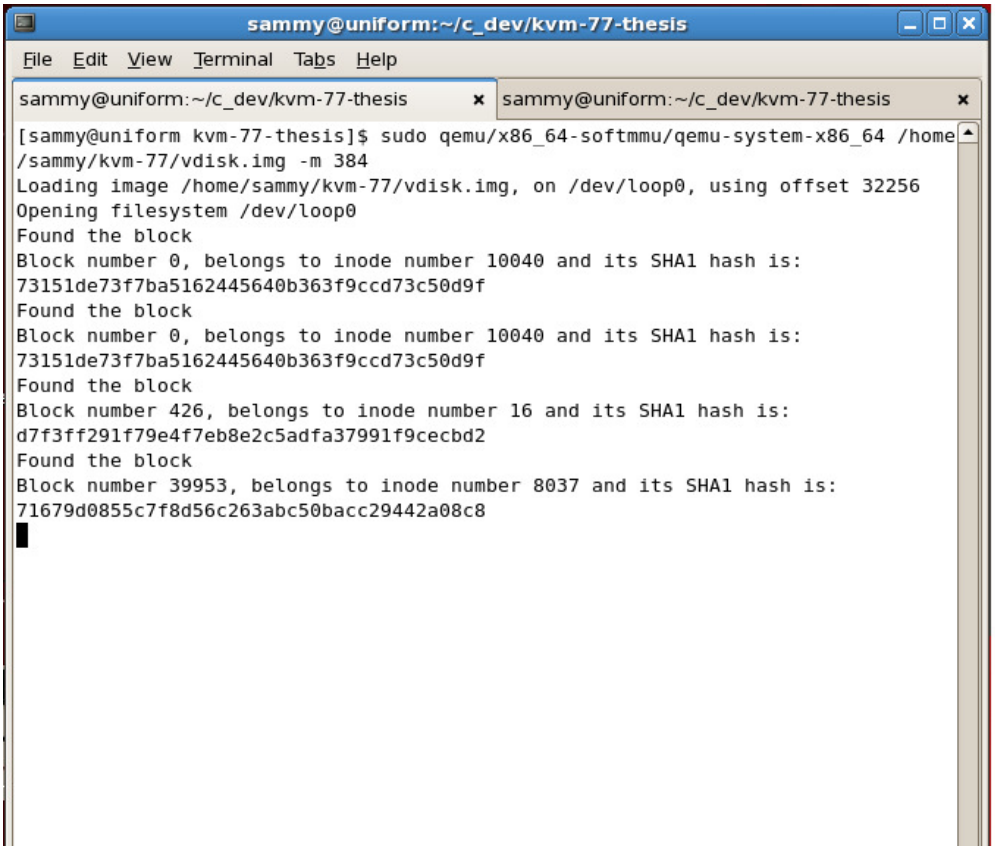

Figure 5-14: Host OS Output

The hash is computed by using the OpenSSL cryptographic library. The shadow filesystem keeps track of which blocks belong to an inode. In order to hash the contents of an inode, the `lseek` and `read` functions are used to read a file from the virtual disk. The call to `lseek` would include the offset into the virtual disk. The offset is calculated by multiplying the sector number (of the file being accessed) and 512 (bytes per sector). The assumption is that data blocks for an inode are contiguous. Even if this assumption is false, all blocks for a given inode are still available in our lookup table.

Once the data of the file are obtained, calls to the OpenSSL EVP API [23] are made to hash them. EVP is a high level interface to the OpenSSL cryptographic functions. Bytes are read from the virtual disk in segments, so the `MD_CTX` (message digest context) structure is used. As bytes are read, they are fed into the `MD_CTX` structure using the `EVP_DigestUpdate` function. After all bytes are read, a call to `EVP_DigestFinal_ex` is made to indicate the end of input and resources are cleaned up using `EVP_MD_CTX_cleanup`. The final computed hash is stored in an allocated character buffer.

Due to the fact that the raw disk format is being used, the raw block driver is used by KVM (`block-raw-posix.c`). This is the file that requires modification to include lookup code to our shadow filesystem. More specifically, the `raw_aio_read` function houses the code which setups up asynchronous access to data on the virtual disk. One of the parameters passed to the `raw_aio_read` function is the sector number needed by our lookup function.

5.5 Building the Lookup Table

Building the lookup table involves using the `libext2fs` library to read the filesystem information. The following section describes how the shadow filesystem is built in the proof of concept. In order to read the virtual disk as a block device, a loop device needs to be created using `losetup` and the virtual disk needs to be attached to it. Once the loop device is created, the libext2fs library can be used on the device as if it was a physical hard disk.

The filesystem is opened using `ext2fs_open`. `libext2fs` uses an abstract structure called `ext2_inode_scan` to scan the filesystem. The `ext2_get_next_inode` function is used to iterate through all inodes of the filesystem. Only inodes that contain valid blocks, and is not a directory is processed. Inodes up to 10 are reserved and 11 is `lost+found`, so they're not included either. Within the iteration loop, a call to `ext2fs_block_iterate` is executed with a function pointer (callback). The pointer refers to a function we define that contains logic to record the block number and map it to the inode currently being processed. Once all inodes are processed, the filesystem could be unloaded, and KVM can resume loading the virtual disk.

5.6 Translating Sectors to Logical Blocks

A block is the most basic unit of a filesystem; on the other hand disks use sectors. Sectors are normally 512 bytes, while our proof of concept use blocks of 1024 bytes. The KVM code uses sectors, and our shadow lookup table uses blocks. In order to make the lookup, a translation needs to be performed. Translating from block numbers to sectors requires the following formula:

$$\mathrm{sec}tor = \left(\frac{(block \times block\_size)}{\mathrm{sec}tor\_size} \right) + \left(\frac{offset}{\mathrm{sec}tor\_size} \right)$$

Translating from sectors to blocks requires the following formula:

$$block = \left(\mathrm{sec}tor - \left(\frac{offset}{\mathrm{sec}tor\_size} \right) \right) \times block\_size$$

The proof of concept handles this translation by providing internal functions `translate_sector_to_block` and `translate_block_to_sector`.

6 Conclusions

From the proof of concept, we are able to determine that observing low level disk activity; high level information can be inferred. If it is known which blocks belong to which files, a shadow filesystem can be built and used to do a reverse lookup based on blocks/sectors being read/written. This would theoretically work on any filesystem that provides APIs for reading the internal data structures. There would have to be a method

of synchronizing changes from the loaded filesystem to the shadow, otherwise data from the shadow filesystem would become stale.

It would also help that the VMM is open source, so we can inject code to do the lookup.

Being able to pull IMA functionality out of the kernel and into the hypervisor allows for VMs to run any OS. This also allows any VM to reliably attest to a verifier.

7 Future Work & Challenges

Since the shadow filesystem is being built when the VM is first loaded and doesn't receive updates during operation, the research is unable to fully shadow the filesystems state during the operation of the VM. Filesystem modifications are kept in memory in the virtual machine, so there's no trivial method of updating the shadow filesystem as the original filesystem changes.

Modifications to the filesystem such as file writes is an area not covered in this research, but some analysis on EXT2 file modification has been performed to prepare future work. The data blocks of a file called `ps.sh` are monitored and the following behavior is observed. The file contains the following code:

```
#!/bin/sh
ls / > /dev/NULL
```

Figure 7-1: `ps.sh` Contents

The file is modified in three different ways to show filesystem behavior. Table 7-1 describes the changes that were made and illustrate the blocks of the file after modification. `1.out` shows the original data blocks for the file (inode `8036`). In `2.out`, the contents of the file changed, but the file size the same (the casing of a word was changed inside the file). It is observed that the data block pointers remained the same. When new content is added to the file, the data block pointers change and a new contiguous group of blocks are allocated to accommodate the new content, as shown in `3.out`. When content is removed from the file, the blocks remain the same (`4.out`). Table 7-1 summarizes the modifications and the effect it had on the data blocks.

Table 7-1: File Modifications Summary

Modification	Change in Blocks	Output File
Original	None	1.out
Changed the file contents (no change in size)	Stayed the same, contents changed	2.out
Contents were added to file	New data blocks were allocated to file to accommodate new size	3.out
Contents removed	Block numbers remain the same, contents changed	4.out

```
Inode: 8036; Block number: 39945
Inode: 8036; Block number: 39946
Inode: 8036; Block number: 39947
Inode: 8036; Block number: 39948
Inode: 8036; Block number: 39949
Inode: 8036; Block number: 39950
Inode: 8036; Block number: 39951
Inode: 8036; Block number: 39952
Inode: 8036; Block number: 39953
Inode: 8036; Block number: 39954
Inode: 8036; Block number: 39955
Inode: 8036; Block number: 39956
Inode: 8036; Block number: 39958
Inode: 8036; Block number: 39959
```

Figure 7-2: 1.out

```
Inode: 8036; Block number: 39945
Inode: 8036; Block number: 39946
Inode: 8036; Block number: 39947
Inode: 8036; Block number: 39948
Inode: 8036; Block number: 39949
Inode: 8036; Block number: 39950
Inode: 8036; Block number: 39951
Inode: 8036; Block number: 39952
Inode: 8036; Block number: 39953
Inode: 8036; Block number: 39954
Inode: 8036; Block number: 39955
Inode: 8036; Block number: 39956
Inode: 8036; Block number: 39958
Inode: 8036; Block number: 39959
```

Figure 7-3: 2.out

```
Inode: 8036; Block number: 39991
Inode: 8036; Block number: 39992
Inode: 8036; Block number: 39993
Inode: 8036; Block number: 39994
Inode: 8036; Block number: 39995
Inode: 8036; Block number: 39996
Inode: 8036; Block number: 39997
Inode: 8036; Block number: 39998
Inode: 8036; Block number: 39999
Inode: 8036; Block number: 40000
Inode: 8036; Block number: 40001
Inode: 8036; Block number: 40002
Inode: 8036; Block number: 40004
Inode: 8036; Block number: 40005
Inode: 8036; Block number: 40006
Inode: 8036; Block number: 40007
Inode: 8036; Block number: 40008
Inode: 8036; Block number: 40009
Inode: 8036; Block number: 40010
Inode: 8036; Block number: 40011
Inode: 8036; Block number: 40012
Inode: 8036; Block number: 40013
Inode: 8036; Block number: 40014
Inode: 8036; Block number: 40015
Inode: 8036; Block number: 40016
Inode: 8036; Block number: 40017
Inode: 8036; Block number: 40018
Inode: 8036; Block number: 40019
Inode: 8036; Block number: 40020
Inode: 8036; Block number: 40021
Inode: 8036; Block number: 40022
Inode: 8036; Block number: 40023
Inode: 8036; Block number: 40024
Inode: 8036; Block number: 40025
```

Figure 7-4: 3.out

```
Inode: 8036; Block number: 39991
Inode: 8036; Block number: 39992
Inode: 8036; Block number: 39993
Inode: 8036; Block number: 39994
Inode: 8036; Block number: 39995
Inode: 8036; Block number: 39996
Inode: 8036; Block number: 39997
Inode: 8036; Block number: 39998
Inode: 8036; Block number: 39999
Inode: 8036; Block number: 40000
Inode: 8036; Block number: 40001
Inode: 8036; Block number: 40002
Inode: 8036; Block number: 40004
Inode: 8036; Block number: 40005
Inode: 8036; Block number: 40006
Inode: 8036; Block number: 40007
Inode: 8036; Block number: 40008
Inode: 8036; Block number: 40009
Inode: 8036; Block number: 40010
Inode: 8036; Block number: 40011
Inode: 8036; Block number: 40012
Inode: 8036; Block number: 40013
Inode: 8036; Block number: 40014
Inode: 8036; Block number: 40015
Inode: 8036; Block number: 40016
Inode: 8036; Block number: 40017
Inode: 8036; Block number: 40018
Inode: 8036; Block number: 40019
Inode: 8036; Block number: 40020
Inode: 8036; Block number: 40021
Inode: 8036; Block number: 40022
Inode: 8036; Block number: 40023
Inode: 8036; Block number: 40024
Inode: 8036; Block number: 40025
```

Figure 7-5: 4.out

A different file modification scenario could occur where the file `/bin/ps` could be removed and replaced with a new one. Files are represented by inodes, and inodes can be referred to by different file names, if one was to remove `/bin/ps` and replace it with another file from a different location, the new `/bin/ps` would have a different inode. If the new file is loaded into memory, it would appear as if a totally different file is loaded. A filename would have to be included to differentiate the two.

Due to the nature of the filesystem, we are not able to easily map file names to the inodes. This is due to the fact that inodes can have multiple file names associated to it.

Another aspect of IMA that isn't part of this research is taking the measurements list, hashing it and storing it in the TPM. This should theoretically be trivial given the availability of the TrouSerS [23] TCG software stack.

Finally, logic is needed for non contiguous data blocks for a given inode.

8 Acknowledgments

The author would like to thank all those who helped complete this work.

- Dr. Charles Border of the NSSA department at RIT; for his guidance and advising throughout the entire thesis process.
- David R. Safford of the T. J. Watson Research Center at IBM; for helping develop the idea for this work.

9 References

The following are text and supporting documents referenced within this paper

[1] T. Garfinkel and M. Rosenblum. "**A Virtual Machine Introspection Based Architecture for Intrusion Detection**". Proc. of the 2003 Network and Distributed System Security Symposium, 2003

[2] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. "**Detecting Past and Present Intrusions through Vulnerability-specific Predicates**". Proc. of the 2005 SOSP, Oct. 2005

[3] X. Jiang, X. Wang, D.Xu. "**Stealthy Malware Detection Through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction**" 2007

[4] R. Sailer, T. Jaeger, X. Zhang, L. van Doorn, "**Attestation-based Policy Enforcement for Remote Access**" in 11th ACM Conference on Computer and Communications Security (CCS), 2004

[5] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, "**Design and Implementation of a TCG-based Integrity Measurement Architecture**" in Proceedings of the 13th USENIX Security Symposium, 2004

[6] R. Meushaw and D. Simard. "**NetTop: Commercial Technology in High Assurance Applications**". Tech Trend Notes: Preview of Tomorrow's Information Technologies, Sept. 2000

[7] Bryan D., Payne Martim D. P. de A. Carbone, Wenke Lee. "**Secure and Flexible Monitoring of Virtual Machines**". 2007

[8] T. Jaeger, R. Sailer, U. Shankar, "**PRIMA: Policy-Reduced Integrity Measurement Architecture**" in ACM Symposium on Access Control Models and Technologies , 2006

[9] K. Kourai and S. Chiba. "**HyperSpector: Virtual Distributed Monitoring environments for Secure Intrusion Detection**" Proc. of the 1st ACM/USENIX international conference on Virtual execution environments, June 2005

[10] High Assurance Platform. http://www.nsa.gov/ia/industry/HAP/HAP.cfm?MenuID=10.2.1.6

[11] VMsafe: A Security Technology for Virtualized Environments. http://www.vmware.com/technology/security/vmsafe.html

[12] XenAccess Library. http://code.google.com/p/xenaccess/

[13] Trusted Computing Group, Trusted Platform Module Specification. https://www.trustedcomputinggroup.org/specs/TPM/

[14] Trusted Computing Group. https://www.trustedcomputinggroup.org

[15] Kernel-based Virtual Machine. http://www.linux-kvm.org/page/Main_Page

[16] The Second Extended Filesystem. http://e2fsprogs.sourceforge.net/ext2.html

[17] Integrity Measurement Architecture. http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_ima.index.html

[18] TrouSerS – the open-source TCG Software Stack. http://trousers.sourceforge.net/

[19] OpenSSL EVP API. http://openssl.org/docs/crypto/evp.html

[20] Design and Implementation of the Second Extended Filesystem. http://e2fsprogs.sourceforge.net/ext2intro.html

10 Appendix A

Attached is the source code required for implementation of a shadow filesystem and changes to the KVM source to support it.

10.1 shadow_fs.c

```
/*
 * shadow filesytem implementation (shadow_fs.c)
 *
 * Created on: January 17, 2009
 *     Author: Sammy Lin <scl9891@cs.rit.edu>
 */

#include "shadow_fs.h"

#define BYTES_PER_SECTOR 512
#define BYTES_PER_BLOCK 1024

ext2_inode_scan scan = NULL;
ext2_filsys fs = NULL;
unsigned int blocks_count;
unsigned int block_size;

/* Our lookup tables */
s_block *blocks_array = NULL;
s_inode *inodes_array = NULL;

/*
 * Build the command for creating the loop device
 * syntax:
 * /sbin/losetup -o offset device image
 * Note: this function is internal only, due to the fact that
 * this function dynamically allocates memory for the strings
 * and requires the caller to free it
 */
static
char
*build_losetup_command(int offset, char *dev, char *image)
{
        char *losetup = "/sbin/losetup -o ";
        char *fullcommand;
        char str_offset[20];
        sprintf(str_offset, "%d", offset);

        /* 3 = two spaces and a trailing null terminator */
        fullcommand = (char *)malloc(strlen(losetup) + (strlen(str_offset) +
                                     strlen(dev) + strlen(image) + 3)
                                     * sizeof(char));
        strcpy(fullcommand, losetup);
        strcat(fullcommand, str_offset);
        strcat(fullcommand, " ");
        strcat(fullcommand, dev);
        strcat(fullcommand, " ");
        strcat(fullcommand, image);

        return fullcommand;
}

/*
 * Load the filesystem and initialize the error table
 *
 */
static
int
load_fs(int offset, char *dev, char *image)
{
       int status = 0;
```

```
        errcode_t err = 0;
        char *losetup_command;

        losetup_command = build_losetup_command(offset, dev, image);
        status = system(losetup_command);
        free(losetup_command);
        if(status){
               printf("Error creating loop device %s, for image %s\n",
                       dev, image);
               return 1;
        }

        initialize_ext2_error_table();

        /* Open the filesystem */
        printf("Opening filesystem %s \n", dev);
        err = ext2fs_open(dev, EXT2_FLAG_JOURNAL_DEV_OK,
                          0, 0, unix_io_manager, &fs);
         if(err){
                 com_err("shadow_fs", err,
                         "while retrying to open fs %s", dev);
                 return 1;
         }
        blocks_count = fs->super->s_blocks_count;
        block_size = fs->blocksize;
#ifdef DEBUG
        printf("Number of blocks: %d, Blocksize: %d \n",
                 blocks_count, block_size);
#endif
        return 0;
}

/*
 * Build the command for deleting the loop device
 * syntax:
 * /sbin/losetup -o /dev/loop0
 * Note: this function is internal only, due to the fact that
 * this function dynamically allocates memory for the strings
 * and requires the caller to free it
 */
static
char
*build_losetup_delete_command(char *dev)
{
         char *losetup = "/sbin/losetup -d ";
         char *fullcommand;

         /* 3 = two spaces and a trailing null terminator */
         fullcommand = (char *)malloc(strlen(losetup) + strlen(dev) + 3
                                      * sizeof(char));
         strcpy(fullcommand, losetup);
         strcat(fullcommand, dev);

        return fullcommand;
}

/*
 * Unload the file system and cleanup used resources
 */
static
int
unload_fs(char *dev)
{
#ifdef DEBUG
        printf("Removing device: %s\n", dev);
#endif
        int status = 0;
        errcode_t err = 0;
        char *losetup_command;

        ext2fs_close_inode_scan(scan);
```

```
        /* Close the filesystem */
        err = ext2fs_close(fs);
        if(err){
                com_err("shadow_fs", err,
                        "while trying to close fs %s", dev);
               return 1;
        }

        /* Delete loop device */
        losetup_command = build_losetup_delete_command(dev);

        status = system(losetup_command);
        free(losetup_command);
        if(status){
                printf("Error removing loop device %s\n", dev);
                return 1;
        }
       return 0;
}

/*
 * Build the pseudo file system (our look-up table)
 */
static
int
build_table()
{
        char *dev;
        /*Initialize the look up tables */
       int num_blocks = 0;
       int num_inodes = 0;

       errcode_t err = 0;

       static s_inode *inodes_arr_iter = NULL;
       static s_block *blocks_arr_iter = NULL;

       num_blocks = fs->super->s_blocks_count;
        num_inodes = fs->super->s_inodes_count;

       blocks_array = (s_block *) malloc(num_blocks * sizeof(s_block) + 1);
        inodes_array = (s_inode *) malloc(num_inodes * sizeof(s_inode));

       dev = (char *) malloc(strlen(fs->device_name) + 1);
       strcpy(dev, fs->device_name);

       inodes_arr_iter = inodes_array;
        blocks_arr_iter = blocks_array;

        /* Start the scan of the inodes of the filesystem */
        err = ext2fs_open_inode_scan(fs, 0, &scan);

        if(err){
                com_err("shadow_fs", err,
                        "while trying to establish a scanner for fs %s",
                        fs->device_name);
                return 1;
        }

        ext2_ino_t inode_num;
        struct ext2_inode inode;

        err = ext2fs_get_next_inode(scan, &inode_num, &inode);
        if(err){
               com_err("shadow_fs", err,
                               "while trying to get next inode for fs %s",
                        fs->device_name);
                return 1;
        }
```

```
        /*
         for each inode, get the inode number, add the inode number to the
         array of inode numbers,
         for each block for that inode, create a s_block struct containing
         the block number and a pointer to inode number for which it belongs
         */
        while( inode_num != 0 ){
                 inodes_arr_iter->inode_num = inode_num;
                inodes_arr_iter->i_size = inode.i_size;
                 inodes_arr_iter->first_block = blocks_arr_iter;
                /* get the filename */
                 blocks_arr_iter->inode_p = inodes_arr_iter;
                        blocks_arr_iter->first = 1; //Make this block the first

                if(ext2fs_inode_has_valid_blocks(&inode) &&
                        ext2fs_check_directory(fs, inode_num) ==
                        EXT2_ET_NO_DIRECTORY &&
                        inode.i_blocks != 0 &&
                          inode_num > 11){
                          //inodes up to 10 are reserved, inode 11 is lost+found
#ifdef DEBUG
                          printf("INODE Number %d, has %d blocks, size: %d:\n",
                                  inode_num, inode.i_blocks, inode.i_size);
#endif
                        //inodes_arr_iter->has_first = 0;
                        ext2fs_block_iterate(fs, inode_num,
                                                BLOCK_FLAG_DATA_ONLY,
                                                NULL, block_iterate_callback,
                                                &blocks_arr_iter);
                }

                inodes_arr_iter++;
                 err = ext2fs_get_next_inode(scan, &inode_num, &inode);
                 if(err){
                          com_err("filesystemviewer", err,
                                  "while trying to get next inode for fs %s",
                                  fs->device_name);
                          return 1;
                 }
         }
        //*inodes_arr_iter = NULL;
        unload_fs(dev);
        return 0;
}


/*
 * Given a block number, and the beginning of the array of blocks,
 * find the inode for which that block belongs
 * Returns 0 if found, 1 if not found
 */
static
int
get_inode_from_block2(const blk_t block_num, s_block *blockp, int len,
                        s_inode **inode)
{
         int i;
         for(i = 0; i < len; i++, blockp++){
                 if(blockp->block_num == block_num){
#ifdef DEBUG
                          printf("Found inode: %u\n",
                                  (blockp->inode_p)->inode_num);
#endif
                        *inode = blockp->inode_p;

                          return 0;
                 }
         }
         return 1;
}
```

```
/*
 * Translate a sector number to a block number
 */
blk_t
translate_sector_to_block(unsigned int sector, s_shadow_fs *sfs)
{
        /*
        To translate a sector to a block, we need to apply the
        following formula:
        block = (sector - offset (in sectors) * 512)/1024
        */
#ifdef DEBUG
        printf("s2b sector: %u, offset=%u\n", sector, sfs->offset);
#endif
        return ((sector - (sfs->offset / BYTES_PER_SECTOR)) *
                BYTES_PER_SECTOR)/BYTES_PER_BLOCK;
}

/*
 * Translate a block number to a sector number
 */
unsigned int
translate_block_to_sector(blk_t block, s_shadow_fs *sfs)
{
        /*
        To translate a block to a sector, we need to apply the
        following formula:
        sector = ((blocks*1024)/512)+(sfs->offset / BYTES_PER_SECTOR)
        */
#ifdef DEBUG
        printf("b2s sector: %u, offset=%u\n", block, sfs->offset);
#endif
        return ((block*BYTES_PER_BLOCK)/BYTES_PER_SECTOR)+
                (sfs->offset/BYTES_PER_SECTOR);
}

/*
 * Check if a sector is first of a file
 */
int
is_first(unsigned int sector_num, s_shadow_fs *sfs)
{
        //translate the sector to block, then find if this block is first in a file
        int i;
        s_block *block_iter;
        blk_t block_num = translate_sector_to_block(sector_num, sfs);
        block_iter = blocks_array;
        for(i = 0; i < blocks_count; i++){
                if(block_iter->block_num == block_num){
                        if(block_iter->first == 1)
                                return 1;
                        else
                                return 0;
                }
                block_iter++;
        }
        return 0;
}

/*
 * Public interface to the shadow file system - Open the filesystem
 */
int
open_shadow_fs(s_shadow_fs *sfs)
{
        load_fs(sfs->offset, sfs->dev, sfs->image);
        build_table();
        return 0;
}

/*
```

```
 * Public interface to the shadow file system - Close the filesystem
 */
int
close_shadow_fs(s_shadow_fs *sfs)
{
        free(inodes_array);
         free(blocks_array);
        return 0;
}

/*
 * Callback used by libext2fs for iterating the blocks of a filesystem
 */
int
block_iterate_callback(ext2_filsys FS, blk_t *BLOCKNR, int BLOCKCNT,
                        void *PRIVATE)
{
        s_block **sblockpp = (s_block**)PRIVATE;
         (*sblockpp)->block_num = *BLOCKNR;
         s_inode *temp_inop = (*sblockpp)->inode_p;
#ifdef DEBUG
        printf("Inode: %u; Block number: %u\n",
                 ((*sblockpp)->inode_p)->inode_num, *BLOCKNR );
#endif
        (*sblockpp)++;
         (*sblockpp)->inode_p = temp_inop;
        (*sblockpp)->first = 0;
         return 0;
}

/*
 * Returns the hash for the inode, this value is dynamically allocated
 * it must be free'd once it's purpose has been fulfilled
 */
int
hash_inode(s_inode const *inode, s_shadow_fs *sfs, unsigned char **hash,
               unsigned int *hash_len)
{

#ifdef DEBUG
        printf("Hashing inode %u\n", inode->inode_num);
#endif
        int             fd;
        unsigned int    sector_num;
        off_t           off;
        unsigned char   buf[BYTES_PER_BLOCK];
        char            *digest = "sha1";
        //unsigned int bytes_rec = 0;
        unsigned int    left_to_read = inode->i_size;
        unsigned int    read_amt; //How much to read
        ssize_t          got;
        /* OpenSSL EVP variables */
        EVP_MD_CTX      mdctx;
        const EVP_MD    *md;
        //unsigned char        md_value[EVP_MAX_MD_SIZE];
        //unsigned char        *md_value;
        *hash = (unsigned char*)malloc(EVP_MAX_MD_SIZE *
                                       sizeof(unsigned char));

        fd = open(sfs->image, O_RDONLY, O_LARGEFILE);
        if(fd == -1){
                printf("Could not open image %s for reading %s\n", sfs->image,
strerror(errno));
                return 1;
        }

        OpenSSL_add_all_digests();
        md = EVP_get_digestbyname(digest);
        EVP_MD_CTX_init(&mdctx);
        EVP_DigestInit_ex(&mdctx, md, NULL);
        //foreach block do hash it
```

```
        //How do we get the blocks of a inode?
        //hash block, block++ while block != first?
        //if the first block is NOT first, report error
        if((inode->first_block)->first != 1){
                printf("Expected first block, but didn't get it\n");
                free(hash);
                return 1;
        }
        s_block *block_iter = inode->first_block;
        //printf("Size of this inode: %u\n", inode->i_size);
        do{
#ifdef DEBUG
                printf("Starting loop\n");
                if( block_iter == NULL){
                        printf("ERROR READING NEXT BLOCK\n");
                }
                printf("Hashing block %u\n", block_iter->block_num);
#endif
                read_amt = (left_to_read > BYTES_PER_BLOCK) ?
                        BYTES_PER_BLOCK : left_to_read;
                /*if( left_to_read > NUM_BYTES)
                        read_amt = NUM_BYTES;
                else
                        read_amt = left_to_read;
                */
                sector_num = translate_block_to_sector(block_iter->block_num,
                                                       sfs);
#ifdef DEBUG
                printf("Sector num being read %u\n", sector_num );
                printf("Going to read %u bytes\n", read_amt);
#endif

                off = lseek(fd, sector_num * BYTES_PER_SECTOR, SEEK_SET);
                got = read(fd, buf, read_amt);

                if(read_amt < BYTES_PER_BLOCK){
                        buf[read_amt] = '\0';
                }

#ifdef DEBUG
                printf("Contents: %s####\nSize: %" PRId64 "\nSupposedly reading %u\n",
                                buf, strlen(buf), read_amt);
                printf("Moved %" PRId64 " bytes from beginning of file\n", off);
                if(got < 0 ){
                        printf("An error occured during reading\n");
                        printf("GOT %" PRId64 " bytes\n", got);
                }
#endif
                //bytes_rec+= strlen(buf);
                left_to_read -= BYTES_PER_BLOCK;
                EVP_DigestUpdate(&mdctx, buf, got);
                block_iter++;
#ifdef DEBUG
                printf("Advancing to next block\n");
                if(block_iter == NULL)
                        printf("Next pointer is NULL\n");
                printf("Next block number: %u first? %d\n",
                                block_iter->block_num, block_iter->first);
                printf("Ending loop\n");
#endif
        }while(block_iter->first != 1);

        close(fd);
        EVP_DigestFinal_ex(&mdctx, *hash, hash_len);
        EVP_MD_CTX_cleanup(&mdctx);

#ifdef DEBUG
        printf("Returning\n");
#endif
        return 0;
}
```

```
/*
 * Get an inode given the block
 */
int
get_inode_from_block(const blk_t block_num, s_inode **inode)
{
        return get_inode_from_block2(block_num, blocks_array,
                                     blocks_count, inode);
}
```

10.2 shadow_fs.h

```
/*
 * Header file for shadow file-system implementation (shadow_fs.h)
 *
 * Created on: Jan 17, 2009
 *     Author: Sammy Lin <scl9891@cs.rit.edu>
 */
#ifndef _SHADOW_FS_H
#define _SHADOW_FS_H
#define _GNU_SOURCE

#include <ext2fs/ext2fs.h>
#include <ext2fs/ext2_fs.h>
#include <ext2fs/ext2_io.h>
#include <et/com_err.h>
#include <ext2fs/ext2_err.h>
#include <openssl/evp.h>
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

struct s_inode;

typedef struct s_block{
        int first;  /* First block in the file? */
        blk_t block_num;
        struct s_inode *inode_p;
}s_block;

typedef struct s_inode{
        ext2_ino_t     inode_num;
       unsigned int   i_size;
       //Used to determine if the first block for this inode has been set
       int            has_first;
        s_block               *first_block;
       /*
        Since an inode can possibly have multiple filenames
        files could point to the "first" file name that represents
        this inode
        */
        //char *files;
}s_inode;

typedef struct s_shadow_fs{
       char *image;
       char *dev;
       int offset;//THIS SHOULD BE A UNSIGNED INT
       int (*get_inode_from_block)(const blk_t block_num, s_inode **inode);
       int (*open_shadow_fs)(struct s_shadow_fs *sfs);
       int (*close_shadow_fs)(struct s_shadow_fs *sfs);
}s_shadow_fs;

extern
int
open_shadow_fs(s_shadow_fs *sfs);

extern
int
is_first(unsigned int sector_num, s_shadow_fs *sfs);

extern
blk_t
translate_sector_to_block(unsigned int sector, s_shadow_fs *sfs);
```

```
extern
unsigned int
translate_block_to_sector(blk_t block, s_shadow_fs *sfs);

extern
int
close_shadow_fs(s_shadow_fs *sfs);

extern
int
get_inode_from_block(const blk_t block_num, s_inode **inode);

extern
int
block_iterate_callback(ext2_filsys FS, blk_t *BLOCKNR, int BLOCKCNT,
                       void *PRIVATE);

extern
int
hash_inode(s_inode const *inode, s_shadow_fs *sfs, unsigned char **hash,
              unsigned int *hash_len);

#endif
```

10.3 Makefile

```
#Compiler macros
CC = gcc
CC_OPTS = -Wall

#Files
C_FILES = shadow_fs.c main.c
#H_FILES = shadow_fs.h
OBJ_FILES = $(SOURCEFILES:.c=.o)
SOURCEFILES = $(C_FILES)
#SOURCEFILES = $(C_FILES) $(H_FILES)
EXECUTABLE = shadow_tester
LIBS = -lext2fs -lcom_err -lcrypto

#Targets

all: clean $(EXECUTABLE)

$(EXECUTABLE): OBJ
	$(CC) $(CC_OPTS) $(DEBUG) $(OBJ_FILES) $(LIBS) -o $(EXECUTABLE)

OBJ:
	$(CC) $(CC_OPTS) $(DEBUG) -c $(SOURCEFILES)

clean:
	rm -rf $(OBJ_FILES) $(EXECUTABLE)

hash: OBJ HASH_OBJ
	$(CC) hashthis.o shadow_fs.o $(LIBS) -o hashthis

HASH_OBJ:
	$(CC) -c hashthis.c

debug:
	$(MAKE) $(MAKEFILE) DEBUG="-DDEBUG"

hash_debug:
	$(MAKE) $(MAKEFILE) hash DEBUG="-DDEBUG"
```

10.4 block-raw-posix.patch

```
23a24
> #include "../shadow_fs/shadow_fs.h"
28a30
> #include <unistd.h>
91c93
<
---
> s_shadow_fs s_fs;
675a678,705
>       if( is_first(sector_num, &s_fs) == 1){
>               blk_t block_num = translate_sector_to_block(sector_num, &s_fs);
>               s_inode *inode;
>               if( inode == NULL){
>                       printf("Could not allocate space for inode\n");
>                       return 1;
>               }
>
>               if(!s_fs.get_inode_from_block(block_num, &inode))
>                       printf( "Found the block\n");
>               else
>                       printf( "Block not found\n");
>
>               printf( "Block number %u, belongs to inode number %u",
>                       block_num, inode->inode_num );
>              printf(" and its SHA1 hash is:\n");
>              unsigned char *hash;
>              unsigned int hash_len;
>              int i;
>              hash_inode(inode, &s_fs, &hash, &hash_len);
>              for(i = 0; i < hash_len; i++)
>                      printf("%02x", hash[i]);
>
>              printf("\n");
>              free(hash);
>       }
```