Hypervisor Integrity Measurement Assistant

Lars Rasmusson[1] and Mazdak Rajabi Nasab[2]
[1]*SICS Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden*
[2]*Chalmers University of Technology, SE-412 96 Gothenburg, Sweden*
Lars.Rasmusson@sics.se, mazdak@student.chalmers.se

Keywords: Cloud computing, hosted computing, security, untrusted code, virtual machine, machine code inspection, checksumming

Abstract: An attacker who has gained access to a computer may want to run arbitrary programs of his choice, and upload or modify configuration files, etc. We can severely restrict the power of the attacker by having a white-list of approved file checksums and a mechanism that prevents the kernel from loading any file with a bad checksum. The check may be placed in the kernel, but that requires a kernel that is prepared for it. The check may also be placed in a hypervisor which intercepts the kernel and prevents the kernel from loading a bad file. Moving the integrity check out from the VM kernel makes it harder for the intruder to bypass the check. We describe the implementation of two systems and give performance results. In the first implementation the checksumming and decision is performed by the hypervisor instead of by the kernel. In the second implementation the kernel computes the checksum and only the final integrity decision is made by the hypervisor. We conclude that it is technically possible to put file integrity control into the hypervisor, both for kernels without and with pre-compiled support for integrity measurement.

1 INTRODUCTION

The techniques for building completely secure system has made significant strides as we have seen first correctness proofs for an OS kernel, for C compilers, and for the gate level implementation of multicore CPUs. However, even if the kernel is correct, there will likely always be bugs in application programs, as they are more diverse, modified much more frequently, and generally built by fewer developers, or built by composing code from different sources (libraries, etc.). Thus, even if the OS kernel is correct, there may still be be opportunities for intruders to take control of a computer via the application programs that run on it. This paper is concerned with how to limit the abilities of an attacker who has broken in to a computer.

The approach taken here is to prevent an attacker who has broken into a computer via an application program from executing arbitrary programs.[1] The computer owner creates in advance a list with checksums for each file that the kernel is allowed to load, and then a mechanism prevents the kernel from loading a file if its checksum does not match with the list. While this mechanism does not prevent the attacker from (mis)using existing programs, the mechanism may, if properly deployed, severely limit the amount of control an attacker can gain over the computer.

A mechanism to prevent loading of unapproved files has recently made its way into the Linux kernel. IMA, the Linux Integrity Measurement Architecture, hooks into the mmap system call inside the kernel and computes the checksum of the entire file just when it is about to be memory mapped. If so configured, IMA forces mmap to fail if the checksum is not correct, and thus prevents a user process to access files with incorrect checksums.

IMA is built into the kernel and executes inside the kernel and therefore only works in kernels that have been prepared with IMA from the beginning. However, in a hosted virtualized environment, such as in a cloud computing infrastructure, it may also be desirable to be able to run arbitrary kernels of the customer's choice, while at the same time be able to protect against intrusion in the customers' VMs.

Another security complication is that the integrity of the checksum list must be protected by the kernel. If the list is kept in the kernel its authenticity can be

[1]In general an attacker has many more options to gain control, including using the programs that are already available. However, not being able to run scripts, etc., severely limits any automated attack.

stated with a with a digital signature. However, the public key used to verify the list's signature must be protected, and it may therefore be desirable to move the key out from the VM and into the hypervisor.

We have therefore developed two prototypes to test different ways to provide hypervisor support for file checksumming and policing (or integrity measurements, in IMA words).

The first prototype is a hypervisor that is able to prevent a Guest kernel from loading bad/tampered files. At similar places where IMA would intercept, the hypervisor intercepts the Guest VM execution near the end of mmap, computes the checksum and makes the mmap call fail if the checksum is incorrect. But, differently from IMA, the first prototype does not require any checksumming code inside the kernel. The entire checksum computation is moved into the hypervisor.

The second prototype relies on the kernel to compute the checksum with IMA, and just checks the integrity of the value in the white-list by verifying the white-list signature with a public key kept in the hypervisor.

The next section describes the implementation of the two approaches. Section 3 reports the results of the performance measurements. In section 4 we discuss the conclusions that can be drawn from the observation. In section 5 we discuss related work which this work is based on, or which addresses a similar problem, and the paper conclusions are given in section 6.

2 HYPERVISOR MEASUREMENT IMPLEMENTATION

2.1 Approach 1: Hypervisor Checksumming

2.1.1 Design

One problem with moving the checksumming out from the kernel is the question of how the hypervisor can get hold of the content in files inside the VM. The hypervisor can't simply read the file contents from the VM's disk. A kernel can manage several different file systems, files can be encrypted on disk or retrieved over the network. Therefore the VMM cannot simply bypass the VM and read the file contents from the Guest's disk before the Guest kernel is allowed to proceed.

Our approach is to let the hypervisor intercept and divert the Guest kernel's execution just before it is about to return from mmap. Then the hypervisor retrieves the file by tricking the kernel into loading the file contents. The hypervisor injects several function calls into the kernel by modifying the VMs program counter (EIP) and registers of the kernel thread that requests the mmap.

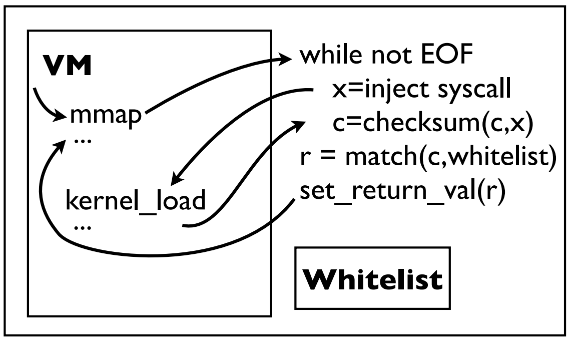

Figure 1: In the first approach, the hypervisor computes the checksum.

First the hypervisor makes the thread call kmalloc to reserves one page of memory inside the kernel's address space. This page is mapped into the hypervisor's memory so the hypervisor can have quick access to the page contents.

Next the hypervisor retrieves the file contents by repeatedly injecting calls to `kernel_read` to load the file, one page at a time, see figure1. For each page that is retrieved, the hypervisor updates the file checksum. Finally it calls kfree to release the memory page, adjusts mmap's return value in register EAX as desired, and restores the EIP to continue the execution as normal.

Our approach requires that the hypervisor is aware of which addresses to intercept, and it also needs information about some of the kernel data structures, in particular the offsets to the fields in the structs.

The required kernel symbols and data structures are listed in table 1. This information can be obtained from the kernel symbol file and header files. Hackers have other methods for finding the kernel layout and data structures in an unknown kernel. [2]

Invoking kernel calls is not side effect free, and thus not completely invisible to the kernel. Just as with IMA, when mmap has finished, the kernel will have loaded the contents of the entire file. This has two effects, the first one being that the initial call to mmap is much slower (since it has to read the entire

[2] See section 5 - How to gather offsets & symbol addresses, in Smashing The Kernel Stack For Fun And Profit, by Sinan 'noir' Eren, Phrack, Volume 0x0b, Issue 0x3c, Phile #0x06 of 0x10, `http://www.phrack.com/issues.html?issue=60&id=6#article`

Kernel symbol	process_measurement kernel_read kfree security_file_mmap kmem_cache_alloc_trace kmalloc_caches __destroy_inode integrity_inode_free
Data structure	struct file struct path struct dentry struct inode struct super_block

Table 1: Kernel symbols and data structures needed by the hypervisor

file), and the second one is that subsequents calls to read are faster, since some of the blocks will be in the page cache.

In Linux without IMA, memory-mapping a file does not load any of the files content. It just installs a handler that detects when the user process tries to read from the file and then loads only the requested pages (by trapping memory page faults). Thus, if only some bytes in the middle of the file are read, only the corresponding block will be loaded from disk.

IMA keeps one checksum for the entire file. This means that when the file is mapped, the entire file needs to be read in for the checksum to be computed. This makes mapping large files slow, and is a substantial way to how Linux currently works.

One can imagine other approaches to checksumming the files, such as having separate checksums for different parts of the file (such as one per disk block, or one per megabyte). These checksums could be checked when the blocks are actually loaded into memory. Then it would not be mmap that fails if the file has a bad checksum. Instead the program would get a memory protection error later, when the virtual memory system has triggered loading a corrupted part of the file.

While the late check is faster, it has its drawbacks. Failing in the middle of an execution may enable situations where an attacker may cause a program to start because the first blocks are unmodified, and then fail later upon accessing a modified part of the file. This may leave the system in an inconsistent state, which may potentially be exploited for attacks.

2.1.2 Implementation

In our implementation we are using a hypervisor which is a modified version of the Bochs x86 emulator. The original Bochs emulator is implemented in C++. Each CPU instruction is implemented as an ”instruction function,” individual method invocations on a CPU object. We have modified the implementation in two ways.

First we have moved out the CPU instructions into a separate file which is compiled into LLVM bit code, for two reasons. One is that the LLVM is a compiler toolkit with an intermediate code representation that is very good for doing optimization passes. We anticipate to be able to speed up execution by optimizing together the sequences of instructions in a basic block. The other reason is that we can add very efficient probes into the instruction functions. A test that would usually require the use of an MMU or CPU debug functionality may be inlined as regular instructions into the native code, or even eliminated completely. The optimization work is not done yet, so the LLVM step still incurs some additional overhead.

The second way we have modified Bochs is to add the integrity mechanism. It adds a test in the CPU instruction loop to see if the EIP (Instruction Pointer) has reached the address that we want to intercept. This check is inefficient and costly, and of the kind that we expect to be able to mostly optimize away with the technique outlined above. If we have reached the monitored address, we create a context record in the hypervisor's memory that stores the kernel thread's current CPU state, including registers, stack pointer and an intercept mode state.

By looking at the contents of the register and traversing the Guest's memory, we can retrieve information from the kernel data structures, such as file pointers, filename, mode bits, file size, etc. If the file should be checksummed according to the policy, we perform a binary search in a list of previously computed hashes to see if the file was already checked. If not, we start the process of checksumming the file.

To checksum a file, we have to go through three modes - allocate - read - free. The kernel thread's current mode is stored in the intercept mode state field in the context record above. If the kernel thread is new, we set it to 'allocate' mode, modify the CPU registers to prepare for a call to kmalloc to allocate one page of kernel memory, and resume computation.

When the CPU returns to the intercepted address, it could either be because kmalloc has returned, or due to another thread calling mmap. We distinguish threads by looking at the value of the stack pointer.

When a thread has returned to the intercepted address from kmalloc, i.e. when it is in the 'allocate' mode, we set the thread's mode to 'read'. We prepare a SHA1 context in the hypervisor memory, and set the CPU registers to call `kernel_read`, to read the first 4k bytes from the file into the Guest memory page that we previously kmalloc:ed, and resume execution. When the same thread eventually returns and is in read state, we update the SHA1 checksum, and if there are more bytes to read, we repeat the call to `kernel_read` for the next 4k bytes, and so on.

When we have reached the end of the file, we set the thread's mode to 'free'. We set the CPU registers to call kfree to free the kernel memory page we allocated earlier, and resume execution. When the thread again reaches the intercepted address and is in free state, we restore the CPU registers from the context record, which is then invalidated, store the computed checksum in the checksum cache, and output it to a log, before we resume the thread's computation.

2.2 Approach 2: Hypervisor Approves Kernel-Computed Checksums

2.2.1 Design

The other approach to hypervisor integrity measurement consists of using the hypervisor's ability to set break points in the Guest VM. In this approach, we have used the Xen hypervisor and the gdbsx functionality which enables the use of gdb to place breakpoints in the Guest VM. In dom0, the debugger, gdb connects to gdbsx which is only availble to the priviledged domain, dom0. The measured domain's memory is made available to dom0 by Xen's ability to map memory pages into multiple domains.

In this approach we have used gdb's scripting capability to run the Guest and execute a python script when a break point is hit. The break point is placed precisely when the IMA code in the Guest kernel is computed. Control is passed back to the gdb in dom0 who checks that the signature is on the approved list, and then resumes execution of the Guest.

As for the first approach, placing the break point requires knowledge of some addresses in the Guest kernel, but fewer, since the Guest execution is never diverted, only suspended. However, in approach 2 we require that the kernel is already compiled with IMA, and the hypervisor only checks that the resulting checksum is indeed a 'trusted' checksum, see figure2.

To improve the security attributes of the system, the VMI application possesses only the public key, and does not have the private key. The private key is only used to sign each file on filesystem on a trusted machine when preparing the Guest VM image. This has impact on the system design. While only the public key is needed for signature verification, the private key is used for signing. Essentially whenever a file is modified its signature has to be updated as well. The set of files the security function will check depends

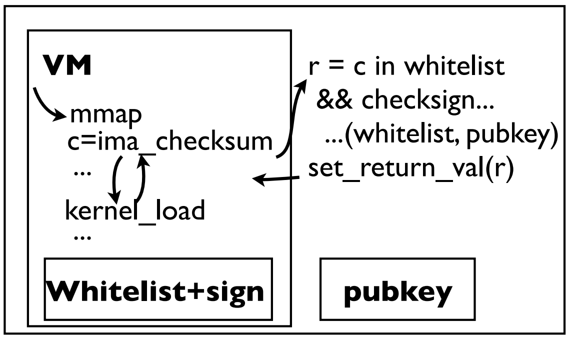

Figure 2: In the second approach, the kernel computes the checksum, and the hypervisor verifies that the whitelist is genuine.

on the security policy and can be configured. Therefore it could be the case that verification only happens for important files, for instance files owned by root, which are expected to be immutable.

2.2.2 Implementation

Kernel debugging is the method used for virtual machine introspection, VMI, in approach 2, and it is already incorporated in the Xen VMM. A python script connects to gdbsx, the gdb server implemented in Xen, and performs the security function. gdbsx provides the functionality for the security function to insert breakpoints at arbitrary addresses in Guest VM's memory. Whenever CPU hits a breakpoint in the Guest VM, it generates a breakpoint exception interrupt. Xen hypervisor controls this interrupt. Xen pauses the Guest VM, and gives control to dom0 which in turn generates a SIGTRAP signal. This signal is handled by gdb, which triggers our python script. Using VMI, the python script in dom0 is notified whenever a file is accessed in the target DomU which verifies the file's signature by the proper public key.

To bridge the semantic gap, some initial information is provided to gdb. To allow gdb to access variables and structures in the Guest VM kernel space, gdb needs to access the compiled directory of the kernel. This allows gdb to know about exact structure of the kernel. Also, the unzipped version of the kernel file has to be available to the gdb. This file contains symbol information of the kernel, and matches information in the system.map file.

To enforce internally denying file access, either a new module has to be installed or an available mechanism in the Linux kernel has to be used. Here we take advantage of the IMA-Appraisal mechanism in the kernel. IMA-Appraisal stores the 160-bit SHA1 digest of files in the extended attribute of a file, "security.ima". In contrast to storing good digest values centrally in a database, saving good SHA1 digests in the file inode removes the search delay. IMA-Appraisal stores valid crypto values in files extended attributes to be enforced internally in a VM. In addition, IMA implements an internal cache which improves the performance.

To handle the problem about updating files' signature, one more security function is designed. The second security function uses only a secret key to verify the HMAC-SHA1 digest of files. HMAC-SHA1 is chosen since this crypto function generates a 160-bit digest, and IMA uses 160-bit SHA1 digest as well. Since these two crypto functions generate 160-bit digests, no kernel modification is required, while in the RSA signature verification, the Linux kernel should be modified to be able to work with 4096-bit RSA signatures. Below the RSA verification design is named VMI-RSA and the HMAC-SHA1 validation design is named VMI-HMAC respectively.

VMI-HMAC is a low-rate context switching security function which validates the HMAC-SHA1 digests of files, in a Guest VM as they are loaded in memory. The security function runs in dom0, and uses a secret key. VMI-RSA is a low-rate context switching security function which verifies the RSA signature of files when they are loaded in the memory of a Guest VM. Verification occurs using the proper public key in dom0. To enhance the security attributes, dom0 and the security function do not possess the private key used for initial signing.

The IMA-Appraisal extended attribute, "security.ima", is 160 bits, while the RSA signature is 4096 bits. For this reason it was decided to design and implement the VMI-HMAC security function that uses the original 160-bit, and does not change the length of security.ima. In this model, VMI in Dom0 possesses a secret key and using that key, it verifies the integrity of files and updates their extended attributes.

IMA uses the `ima_calc_hash` function for calculating the SHA1 digests. This function is only called by `ima_collect_measurement`. To verify the integrity of files VMI-HMAC intercepts `ima_calc_hash` function flow. It is possible to replace the calculated SHA1 digest by inserting a breakpoint in either where `ima_calc_hash` returns or where it is called. We chose to put it in `ima_collect_measurement`, which calls `ima_calc_hash`.

By inserting a breakpoint after where the ima_calc_hash function is called, it is possible to replace the file hash digest, stored in `iint->ima_xattr.digest` with the HMAC-SHA1 digest. The HMAC-SHA1 digest is computed by the VMI application with the secret key. When CPU executes a breakpoint it gives control to the breakpoints handler which is gdb/gdbsx, and then finally a python script is executed, which first reads the SHA1 digest computed by IMA via `iint->ima_xattr.digest`. Next it computes HMAC value based on the secret key and the SHA1 digest, and finally it replaces the SHA1 digest with the newly computed HMAC-SHA1 digest.

For VMI-RSA we needed to modify `IMA_DIGEST_SIZE` in ima.h and `evm_ima_xattr_data.digest` in integrity.h to hold 4096 bits. These changes do not affect IMA normal behavior since all variable and structure fields are initialized with 0. Since VMI-RSA does not have access to the private key it cannot update file checksums. But since it has the public key, it can verify the RSA signature stored in extended security attributes, by intercepting `ima_appraise_measurement` just after it has compared the checksum with the value stored in the security.ima attribute. IMA simply performs a byte-by-byte comparison with the security attribute. VMI-RSA on the other hand will copy over checksum and the 512 bytes of the security attribute into dom0 and verify that it contains a valid signature for the checksum. VMI-RSA then overwrites the return value of the IMA's byte-by-byte comparison to instead report success if and only if the signature was valid.

3 RESULTS

3.1 Approach 1: Hypervisor Checksumming

Here we report on the experiments that were made to study the overhead caused by the hypervisor integrity measurements.

Since the experimental setup is quite complex with layers of software executing on top of others, we performed experiments to first determine how much each layer added to the total execution time. This allows us to finally determine the cost of the integrity measurements themselves.

In the experiment we test the effect of our modifications to the task of memory mapping a single block of medium sized and large file. The effect that is expected is that without integrity measurements, the execution time is very quick, even if the files are large, but with integrity measurements, either by IMA, the hypervisor, or both, the execution time should be considerably longer for the first access of the file. Later accesses to the file should be faster since the checksums are only computed the first time the file is accessed.

The test machine is a Dell R200, quad-core Intel Xeon X3360, 2.83GHz with 4 GiB memory, running Ubuntu 10.04 with Linux kernel 2.6.32. The Guest VM gets 1 GiB of RAM.

In the first experiment we first read one block from five files of size 100 MiB and report the execution time. Then we repeat that experiment 99 times and report the total execution time.

In the second experiment we first read one block from 20 files of size 1 MiB and report the execution time. Then we repeat that experiment 99 times and report the total execution time.

Each of the experiment was repeated three times for each system configuration. The OS in the VM was rebooted between each run to clean the VM page caches.

Four systems were investigated: chroot with the loopback mounted file system, Linux kvm, a regular Bochs without any modifications, a Bochs with the LLVM JIT-compiled byte code, and a Bochs with the LLVM JIT-compiled byte code that performs integrity measurements of the Guest VM.

Each of the four systems were run with Linux IMA either off or on, except for chroot which was only run with IMA off, because IMA support was lacking in the Host kernel.

The time measurements are not very accurate for low values. This is because the VM Guest clocks drifted very much, and we synchronized the local clock with NTP before measuring the start and stop time. This incurred an additive cost of about two seconds, with some second of variance due to network traffic. This is unimportant for understanding the qualitative results of our experiments, and more precision in these measurements will not add to the understanding of the integrity measurement system we report here.

The results are shown in table 2.

Hypervisor	Integrity check	5 files 100 MiB		20 files 1 MiB	
		First access	Next 99 accesses	First access	Next 99 accesses
1. chroot		3	3	3	4
		3	4	3	5
		2	3	2	5
2. kvm		2	4	3	9
		3	5	3	8
		3	5	3	8
3. kvm	IMA	14	4	3	9
		14	4	3	9
		15	5	3	9
4. c++ bochs		1	25	1	74
		1	24	2	72
		1	24	1	73
5. c++ bochs	IMA	675	23	15	68
		678	23	15	67
		677	23	15	67
6. llvm-bochs		1	29	2	90
		1	30	2	89
		1	29	2	88
7. llvm-bochs	IMA	847	28	19	85
		840	30	19	82
		838	27	19	83
8. llvm-bochs	hypervisor	48	31	3	90
		47	30	3	95
		45	30	3	95
9. llvm-bochs	IMA + hypervisor	904	30	21	91
		907	30	20	96
		923	31	21	95

Table 2: The table shows the outcomes of three repeated experiments. Execution time in seconds, including NTP overhead.

3.2 Approach 2: Hypervisor Approves Kernel-Computed Checksums

Here we report on the experiments that were made to study the overhead caused by the hypervisor approving but not computing the file checksums. The values are in table 3.

To determine the sources of execution overhead for approach 2 we run a set of experiments, similarly to approach 1. We vary the number of files and file sizes, and in each case we read just the first 10 KiB of each file in the set using dd. The experiment is run with or without IMA without any virtualization (”bare metal”), in Xen dom0 with direct device access, in domU with indirect device access. Finally we run the VMI-HMAC and VMI-RSA, and a dummy VMI that does nothing but trapping to dom0 and return. The latter is used to estimate the cost of the trapping. As for the approach 1 experiments we use ntp to synchronize the clocks before and after each round of the experiment to mitigate clock drift.

The tests are run on an Intel Core i7-2829QM 2.30GHz processor, with Intel VT, 16GB RAM, and 7200 RPM iSCSI disk. The DomU has 12GB RAM. Fedora 16, 64-bit, is used as OS for all system configurations. The Linux kernel is 3.2.0-rc1 with the IMA-Appraisal patches.

	Test 1 100 files 10 MB each Total 1GB		Test 2 20,000 files 52KB each Total 1GB		Test 3 100 files 100 MB each Total 10GB		Test 4 20,000 files 520KB each Total 10GB	
	1st	Ave.	1st	Ave.	1st	Ave.	1st	Ave.
Bare Metal	*0.260*	*0.046*	*22.033*	*11.123*	0.332	0.046	34.504	11.123
Bare Metal with IMA	*18.398*	*0.047*	*1034.534*	*11.124*	106.386	0.047	1110.298	12.186
Dom0	*0.252*	*0.059*	*28.827*	*13.460*	0.347	0.060	39.143	13.467
Dom0 with IMA	*18.643*	*0.144*	*1098.977*	*46.854*	106.872	0.144	1175.865	47.495
DomU	*0.348*	*0.059*	*31.873*	*13.450*	0.506	0.059	49.075	13.462
DomU With IMA	*15.158*	*0.060*	*86.159*	*13.601*	98.002	0.066	199.328	30.183
VMI Breakpoint	*15.522*	*0.064*	*911.183*	*14.338*	117.502	0.097	931.106	14.384
VMI HMAC	*20.980*	*0.135*	*1985.688*	*30.713*	120.916	0.206	2010.153	30.977
VMI RSA	*23.995*	*0.109*	*2402.922*	*24.750*	124.306	0.109	2449.640	25.085

Table 3: Performance results for approach 2; the execution time measured in seconds. The average is taken over the last five of six runs.

4 DISCUSSION

4.1 Approach 1: Hypervisor Checksumming

Comparing the execution times of the tests in a chrooted environment and a kvm without IMA, we see as expected that they are about as big for the first access. The only significant cost here is the cost of the ntpd synchronization which is performed before and after each synchronization, and it takes approximately (3+3+2)/3 = 2.7 seconds. After removing the time for ntp synchronization removed from the next access times and divide by 500, we can see that each of the next 99 access to the 5 large files takes about 1 ms each for chroot, and also 1 ms for the 20 small files. Kvm takes 4 ms per file for the large files and 3 ms per file for the small files.

The third line in table 2 shows that running a kernel with IMA enabled in kvm significantly increases execution time in particular for the first access to the file. Subsequent accesses are not significantly affected. In the first access, we read 5*100 MiB, so the cost is 23 ms per MiB for the large files. For the small files the same cost would add 0.5 seconds to the values, which is within the measurement error.

The fourth line shows the results of running in an unmodified bochs emulator. In the experiments the clock inside the emulator drifted a lot, which is what forced us to use ntp to set the clocks. This means that the errors are larger for the bochs measurements. For instance, the measurement scripts reported about one second for the first accesses to both large and small files. Since the clocks are synchronized with ntp before and after the experiment the error due to drift is additive rather than multiplicative. Each file access takes around 45 ms and 35 ms for the large and small files respectively in the experiment. The difference may be partially explained as a measurement error, due to a 3 second clock drift per measurement.

Comparing line 4 (c++ bochs) and line 2 (kvm), we see that c++ bochs is about five to seven times slower than kvm for our workload. Comparing line 4 (c++ bochs) and 6 (llvm-bochs), and comparing line 5 (c++ bochs IMA) with line 7 (llvm-bochs IMA) we see that the llvm-bochs is about 20-25 percent slower than c++ bochs in all cases.

The overhead of running IMA can be seen in line 5. The initial access is very slow since IMA has to load and compute the checksum of the entire file. The cost is 1.34 seconds per MB for the large files and only 0.12 seconds for the small files. The difference may be due to the the Host OS being able to cache the small files, implying that with bochs, a cache miss in the Host incurs a ten-fold increase in loading and checksumming. The next 99 accesses are faster than their non-IMA counterparts. This can be explained by recalling that IMA has placed the file content in the Guest OS's page cache, and thus do not have to retrieve it from the Host OS.

We are now in the position of assessing the overhead of putting the integrity check into the hypervisor. Line 8 shows that the hypervisor integrity file is much faster (47 and 3 seconds for large and small files respectively) when compared to having checksum in side the kernel (676 and 15 seconds) when using the bochs hypervisor.

Comparing line 8 with line 6 for the next 99 accesses where no integrity check is made, we see that the hypervisor integrity check code adds 6 percent to the execution time. Comparing line 9 with line 7, we see that the hypervisor integrity check adds 8 percent to the execution time,

From the above we find that the costs for the llvm-bochs hypervisor integrity check can be attributed in most part to the use of bochs as hypervisor, since it incurs a five to seven times execution overhead over kvm. To that is added a 20 percent cost due to our use of a modified bochs, llvm-bochs. On top of that is the 8 percent overhead from the hypervisor integrity check added to arrive from the values in the "next 99 accesses" columns on line 2 to the values on line 8.

The values in "First access column" in line 8 (47 seconds) is the result of reading and checksumming the large files. This operation added about 12 seconds to kvm IMA on line 3, making us expect a value of 90 seconds. We currently do not have a good explanation for why we see a lower value.

We also see a large dip in performance when running IMA inside any of the bochs hypervisors. This dip was initially thought to be due to bad interaction between IMA, bochs and the Host OS, but since the hypervisor on line 8 is also loading the pages via the Guest kernel, it must be a bad interaction between IMA and bochs only.

4.2 Approach 2: Hypervisor Approves Kernel-Computed Checksums

The measured execution time for the second approach provide some insight into where the costs are. In this approach we run most of the Guest code natively in domU using the Intel VT, and only invoke the integrity functions in dom0 when the break point in `ima_calc_hash` is reached.

The two big sources of overhead are the kernel's and IMA's bookkeeping when opening files, and the actual work in computing the checksum. To estimate these individual costs we chose workloads that stressed these two sources individually, by having few or many files, and a small or large file set to checksum. The execution times can be seen in table 3.

To tease out the individual components, we do a least-squares fit of the four values for the length of first execution on each line to

$$time = k_1/1000 * \#files + k_2 * gigabytes$$

where k_1 is the time (in milliseconds) to handle each file, and k_2 is the time to handle 1 GB of data. This fit is very coarse, but the results in table 4 show that IMA adds about 10 seconds per GiB that gets checksummed.

The bookkeeping cost per file rises from 1-2 ms to 50 ms with IMA, which is particularly costly when there are many files that need to be checksummed. HMAC adds another 50 ms per first file access, and RSA adds yet another 20 ms to that, reaching almost 120 ms per first file access. Subsequent file accesses do not reach the break point as the file is marked as already checksummed by the IMA code in the Linux kernel.

Hypervisor		**k_1, ms/file**	**k_2, s/GiB**
bare metal		1.2994	0.41262
	IMA	50.965	9.6601
dom0		1.6046	0.34258
	IMA	54.203	9.7068
domU		1.8657	0.5719
	IMA	4.2216	10.6
breakpoint	IMA	43.668	8.7023
HMAC	IMA	97.505	8.7085
RSA	IMA	118.7	9.5136

Table 4: Least squares fit of the first access to a file to the function $time = k_1/1000 * A + k_2 * B$ where A is the number of files accessed, 100 or 20000, and B is the total size of the set in GB, 1 or 10.

We note that the estimated time per GiB appears to have decreased for the last three lines in table 4. This may indicate that the least squares fit to the data was not very accurate, and some of the cost due to checksumming was incorrectly attributed to the triggering of the breakpoint. However, a re-fit of the data with k_2 set to 10.6 seconds does not significantly change the result, only reducing the least squares estimated cost per file with 0.5 ms.

Unintuitively, the cost per file is very low for the DomU with IMA. In table 3 the execution time for DomU IMA in test 2 is only 86 seconds, compared to 1034 seconds for bare metal IMA. This cannot be explained by caching effects, since the DomU was running on a separate disk device. Further discussion about this anomaly can be found in (Nasab, 2012).

5 RELATED WORK

For virtualization, we have used Bochs (Mihocka and Shwartsman, 2008), which is an emulator written in C++ that can run unmodified kernels on any CPU, and Xen (Barham et al., 2003), which can use either modified kernels (paravirtualization), or take advantage of special virtualization support in CPUs that have it (HVM). By itself Xen only monitors coarse VM behavior such as what goes in and out via the virtual devices, memory and CPU utilization rate, not fine details such as file content.

Virtualization has been used to lock down Guests in many ways. Logging and replay techniques were proposed in (Dunlap et al., 2002). VM introspection for intrusion detection was proposed in (Garfinkel and Rosenblum, 2003; Riley et al., 2008). XenAccess (Payne et al., 2007) enable introspection of a Xen VM without being able to interrupt it, which is why we needed to use gdb-sx instead for our experiments.

iRODS (Wan et al., 2009) is a policy based system for controling cloud clients' access to data in the Amazon S3 file system. iRODS enforces policies at specific enforement points in user programs, not for general files opened by the kernel.

While we are interested in moving IMA-like file integrity checks into the hypervisor, the work by Christodorescu et al. (Christodorescu et al., 2009) use introspection for verifying the integrity of the kernel to protect against root kits. SecVisor (Seshadri et al., 2007) use hardware support to achieve the same thing.

Other approaches focus on preventing the kernel to only load authorized code (Riley et al., 2008; Litty and Lie, 2006). That fails to address the issue with modified config files.

In approach 1 we use machine code to machine code JIT translation to speed up execution. It was used early in the Dynamo system (Bala et al., 2000). PIN (Reddi et al., 2004) and DynamoRIO (Bruening, 2004) are tools that let a user write probes that are dynamically injected into an application level program, not a full OS.

LLVM (Lattner and Adve, 2004) is a compiler framework that has an intermediate code representation that enables programmatical modification, optimization and JIT compilation at runtime. It is used in the Binary Code Inliner to produce optimized code for each translated basic block on the fly.

This work is relevant to the area of cloud computing. Good surveys of multi-tenant cloud risks include (Rodero-Merino et al., 2012; Vaquero et al., 2011; Constandache et al., 2008; Descher et al., 2009; Baldwin et al., 2009). The proposed solution addresses the situation where one needs efficient and automatic methods for detecting and stopping hacked servers from being controlled by intruders.

6 CONCLUSION

We conclude that it is technically possible to put file integrity control into the hypervisor, both for kernels without and with pre-compiled support for integrity measurement. Both techniques require intercepting the kernel at the right place and accessing the Guest VM RAM.

While the presented work has not focused on optimizing the execution time, we have still investigated the sources of execution overhead for two different ways to make a hypervisor enforce file integrity on Guest virtual machines. In the first approach the hypervisor injected calls into the Guest kernel to load the file when the file was first opened, and the hypervisor computed the file checksum. In the second approach, the hypervisor relied on the kernel to compute the checksum, and only matched the checksum with the white-list.

The implementation of the first approach used a modified emulator, which made it easy to intercept the Guest at addesses extracted from the kernel symbol file System.map. The main result is that the overhead of the injection approach is about eight percent for a small file set, after the other sources of overhead have been accounted for.

The use of the emulator was the major source of overhead, responsible for a five time slowdown compared to HVM virtualization via KVM. This overhead may be significantly reduced by changing virtualization technique. An additional 20 percent overhead came from the use of a JIT tool chain to produce the code. This overhead may be removed by using a non-JITed emulator, but the reason to include the JIT in the first place is that it makes possible emulator optimization such as run-time optimizing hot traces, though it was not implemented in the current prototype.

The second approach used Xen's gdb debug facilities to attach python scripts to monitor a Guest VM. We observed that triggering the break point costs around 40 ms per triggering, but the number of triggerings is reduced by Guest kernel only computes the checksum when the file is not in the file checksum cache. The overhead of the approach is between 20 percent for a small file set and 100 percent for a large file set.

REFERENCES

Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA. ACM. http://doi.acm.org/10.1145/349299.349303.

Baldwin, A., Dalton, C., Shiu, S., Kostienko, K., and Rajpoot, Q. (2009). Providing secure services for a virtual infrastructure. *SIGOPS Oper. Syst. Rev.*, 43:44–51. http://doi.acm.org/10.1145/1496909.1496919.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177. http://doi.acm.org/10.1145/1165389.945462.

Bruening, D. L. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.7639.

Christodorescu, M., Sailer, R., Schales, D. L., Sgandurra, D., and Zamboni, D. (2009). Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 97–102, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1655008.1655022.

Constandache, I., Yumerefendi, A., and Chase, J. (2008). Secure control of portable images in a virtual computing utility. In *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '08, pages 1–8, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1456482.1456484.

Descher, M., Masser, P., Feilhauer, T., Tjoa, A. M., and Huemer, D. (2009). Retaining data control to the client in infrastructure clouds. *Availability, Reliability and Security, International Conference on*, 0:9–16. http://doi.ieeecomputersociety.org/10.1109/ARES.2009.78.

Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. (2002). Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In Culler, D. E. and Druschel, P., editors, *OSDI*. USENIX Association. http://www.usenix.org/events/osdi02/tech/dunlap.html.

Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206. http://suif.stanford.edu/papers/vmi-ndss03.pdf.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society. http://llvm.org/pubs/2003-09-30-LifelongOptimizationTR.pdf.

Litty, L. and Lie, D. (2006). Manitou: a layer-below approach to fighting malware. In Torrellas, J., editor, *ASID*, pages 6–11. ACM. http://doi.acm.org/10.1145/1181309.1181311.

Mihocka, D. and Shwartsman, S. (2008). Virtualization without direct execution or jitting - designing a portable vm. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation*. http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf.

Nasab, M. R. (2012). Security functions for virtual machines via introspection. Master's thesis, Chalmers University, Sweden. http://publications.lib.chalmers.se/records/fulltext/160810.pdf.

Payne, B. D., Carbone, M., and Lee, W. (2007). Secure and Flexible Monitoring of Virtual Machines. *Computer Security Applications Conference, Annual*, 0:385–397. http://doi.ieeecomputersociety.org/10.1109/ACSAC.2007.10.

Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 workshop on Computer Architecture Education: held in conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1275571.1275600.

Riley, R., Jiang, X., and Xu, D. (2008). Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Lippmann, R., Kirda, E., and Trachtenberg, A., editors, *RAID*, volume 5230 of *Lecture Notes in Computer Science*, pages 1–20. Springer.

Rodero-Merino, L., Vaquero, L. M., Caron, E., Muresan, A., and Desprez, F. (2012). Building safe paas clouds: A survey on security in multitenant software platforms. *Computers & Security*, 31(1):96 – 108. http://dx.doi.org/10.1016/j.cose.2011.10.006.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In Bressoud, T. C. and Kaashoek, M. F., editors, *SOSP*, pages 335–350. ACM. http://doi.acm.org/10.1145/1294261.1294294.

Vaquero, L. M., Rodero-Merino, L., and Morán, D. (2011). Locking the sky: a survey on IaaS cloud security. *Computing*, 91:93–118. http://dx.doi.org/10.1007/s00607-010-0140-x.

Wan, M., Moore, R., and Rajasekar, A. (2009). Integration of cloud storage with data grids. *Computing*, (October). https://www.irods.org/pubs/DICE_icvci3_mainpaper_pub-0910.pdf.