Runtime Attestation for IAAS Clouds

Jesse Elwell[1], Angelo Sapello[1], Alexander Polylisher[1], Giovanni Di Crescenzo[1], Abhrajit Ghosh[1], Ayumu Kubota[2] and Takashi Matsunaka[2]
[1]*Vencore Labs, Basking Ridge, N.J., U.S.A.*
[2]*KDDI Research, Saitama, Japan*

Keywords: Security, Virtualization, Cloud Infrastructure, Infrastructure-as-a-Service (IAAS).

Abstract: We present the RIC (*R*untime Attestation for *I*aas *C*louds) system which uses timing-based attestation to verify the integrity of a running Xen Hypervisor as well as the guest virtual machines running on top of it. As part of the RIC system we present a novel attestation technique which includes not only the guest operating system's static code and read-only data sections but also the guest OS' dynamically loadable kernel modules. These attestations are conducted periodically at run-time to provide a stronger guarantee of correctness than that offered by load-time verification techniques. A system such as RIC can be used in cloud computing scenarios to verify the environment in which the cloud services ultimately run. Furthermore we offer a method to decrease the performance impact that this process has on the virtual machines that run the cloud services since these services often have very strict performance and availability requirements. This scheme effectively extends the root of trust on the cloud machines from the Xen hypervisor upward to include the guest OS that runs within each virtual machine. This work represents an important step towards secure cloud computing platforms which can help cloud providers offer new services that require higher levels of security than are possible in cloud data centers today.

1 INTRODUCTION

In recent years computer systems have seen a substantial increase in the number of attacks performed against them. One reason for this increase is that the ubiquitous use of these systems for activities such as banking and e-commerce, handling business and/or military secrets, processing medical records, etc. offers an environment where an attacker who manages to successfully compromise the security of these systems can find it quite lucrative. According to (Symantec, 2015) there was a 23% increase in the number of breaches in 2014 when compared to 2013.

Recently there has been a growing interest in a technique known as *attestation* to help protect systems against modification by an attacker. Attestation is a process in which software that runs on a system is measured (verified) to provide assurance of its integrity to other entities. In the most common case the software that is critical to the secure operation of a given system, for example the operating system, is measured and it's integrity is attested to another system.

While the communication portion of different attestation systems are similar, there are different types of verification. Among the most popular are hardware-based verification which is typically supported by the Trusted Platform Module (TPM) (tpm, 2007) and timing-based verification which is implemented in software (Seshadri et al., 2005). Hardware-based verification leverages the fact that a dedicated hardware agent exists to perform measurements (e.g. the TPM) that attackers cannot tamper with to influence the measurement process. Timing-based verification, as its name suggests, relies on the amount of time taken to perform a measurement to detect attacks that attempt to change the measurement process. In this paper we are primarily concerned with timing-based verification although some of the ideas presented here may also be applicable to hardware-based verification. The relative merits of the two approaches are discussed in (Seshadri et al., 2005) and (Ghosh et al., 2014).

Verification can be performed at different times to handle different types of attacks. For example the hardware-based verification supported by the TPM performs verification once at load-time and is capable of detecting modifications made to the binary either

Elwell, J., Sapello, A., Poylisher, A., Di Crescenzo, G., Ghosh, A., Kubota, A. and Matsunaka, T.
Runtime Attestation for IAAS Clouds.
DOI: 10.5220/0006804002330247
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 233-247
ISBN: 978-989-758-295-0

before load time (e.g. on disk) or during the loading process. When compared to load-time verification, run-time verification offers additional protection against attacks that modify software while it runs, for example if an attacker adds a malicious kernel module to a running OS. Run-time verification however requires some extra care to handle some of the differences that can arise even when running the exact same binary (kernel) on two different machines. For example dynamic (run-time) linking can place loadable kernel modules at different addresses, leading to differing jump/branch targets and references to external variables. The RIC system solves this problem allowing the system to perform periodic run-time verification of the Xen hypervisor and the guest operating system's kernel and kernel modules in a manner that accounts for these address changes.

Attestation in any form can add a non-trivial amount of overhead to a running system. For load-time attestation this overhead comes from the fact that measurement must take place before the software can be run. Run-time attestation on the other hand can require the software being measured to be paused prior to and during measurement. For long running programs the overhead of performing a single load-time attestation is negligible, however run-time verification is more costly even for such programs since they are measured periodically throughout their lifetime. The principal component of this overhead usually arises from the need to measure possibly large areas of memory.

Existing work has presented the basic primitives necessary to perform timing-based verification (Seshadri et al., 2005) and to apply timing-based verification to a running Xen hypervisor's kernel for the purpose of securing virtual machines (VMs) running cloud services and operating above Xen (Ghosh et al., 2014). In this paper we assume the XSWAT system proposed in (Ghosh et al., 2014) as our baseline system and present techniques to allow such systems to measure more of the critical software involved in supporting cloud services including the guest VM kernels and their corresponding modules. An overview of the RIC system is depicted in Figure 1. This effectively extends the system's root of trust to include the guest operating system's kernel and modules. Due to the increased overhead from measuring the guest VM's kernel and modules in addition to the Xen hypervisor during each attestation we also present a technique for parallelizing the hashing component of attestation to minimize the performance impact that this technique has on the software that runs on the system.

Owing to a lack of Ethernet support in Xen, XSWAT was devised to attest over a serial link. The use of serial communication links for attestation on a large scale is expected to be infeasible owing to the additional costs involved. To investigate the feasability of performing attestation over available Ethernet infrastructure, a specialized PCI Ethernet card driver was developed for RIC and its performance studied.

The rest of the paper is organized as follows: Section 2 introduces the threat model and assumptions used throughout this work, Section 3 contains details about profiling the operating systems of guest VMs in preparation for run-time attestation which is described in Section 4, Section 5 explains our work in speeding up hashing using parallelism, Section 6 describes the Ethernet driver developed for RIC, Section 7 presents the experimental results collected from the RIC system, Section 8 discusses related work and Section 9 offers concluding remarks. Finally, a formal proof of the security of the parallel hashing scheme presented in Section 5 is offered in Appendix 1.

2 THREAT MODEL & ASSUMPTIONS

The threat model used in this work is similar to that of that of the XSWAT system with a notable exception. While the XSWAT system assumes that an attacker has network access to and can compromise guest OSes (including the privileged Domain 0), the system discussed in this work is designed to detect these guest OS compromises. As such we assume only that the attacker has network access to the cloud machine. RIC is designed to detect attacks that compromise a guest VM's OS. This in turn helps to protect the hypervisor by detecting attacks such as VM escape (Wojtczuk, 2008) that rely on modifying a guest VM's OS to launch an attack against the hypervisor. Effectively, the RIC system limits an attacker's ability to place attack code in any software layer other than the guest VM's user-space.A malicious guest OS can still be used to attack the hypervisor, however to avoid detection the attacker cannot rely on modifying the guest VM's OS kernel to do so. For example, the attacker could utilize a Return Oriented Programming (ROP) (Shacham, 2007) attack against the guest OS to execute code with OS privilege to launch an attack against the hypervisor. Attacks such as ROP attacks launched against the guest VMs or the hypervisor are not covered in our threat model, detection and/or mitigation of such attacks is left for future work.

The proposed system also defends against multiprocessor attacks albeit in a different way than the XSWAT system. Rather than halting all but the bootstrap processor during an attestation request, the pre-

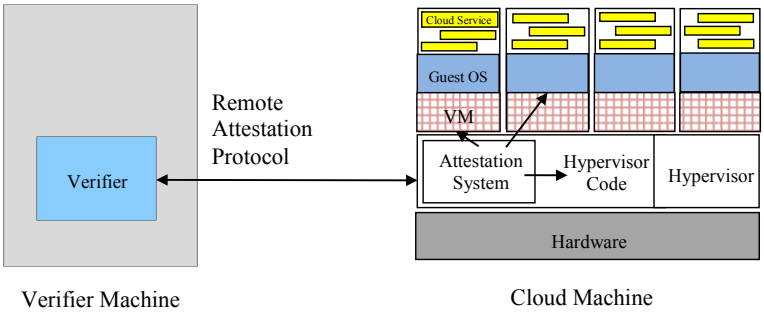

Figure 1: RIC Attestation System Overview.

viously unused processors are now utilized to speed up the hash computation. This means that those processors continue to be unavailable for use by an attacker which is guaranteed and verified as part of the attestation protocol described in Section 4.

As in (Seshadri et al., 2005; Ghosh et al., 2014) we assume that the attacker does not have physical access to the cloud machine and thus cannot tamper with the hardware for example by replacing the CPU with a faster one. We believe that this is a reasonable assumption to make since cloud providers presumably have physical security measures in place in the interest of protecting their reputation as a secure cloud provider.

The process of profiling VMs, described in Section 3, presents a race condition in that an attestation request that completes successfully attests that the VM kernel's code and read-only data have not been modified since it was profiled. However if an attacker is able to modify the VM's kernel code or read-only data *before* the VM is profiled then subsequent verification requests will succeed even though the attack code may already be in place. Therefore we assume that the VMs are profiled before the attacker gains access to them, for example by profiling VMs before the cloud machine is connected to a public network.

Attacks that only require access to the user-level application that offers a cloud service are not covered in our threat model. An example of such an attack would be an SQL injection attack launched against a website that uses an SQL database as a backend to store data.

3 PROFILING A VIRTUAL MACHINE'S OPERATING SYSTEM

In this section we provide a detailed explanation of the improvements made to the XSWAT system that allow a guest VM's OS kernel and modules to be measured in addition to the Xen hypervisor's kernel.

In order to measure (i.e. hash) a running OS, a strategy is needed to hash the various parts of that OS. Similarly to the measurement of the hypervisor, the sections of the OS that can be reliably measured are the code and read-only data sections. Measuring the code and read-only sections of the kernel proper is complicated by the existence of structures such as SMP locks and jump tables. Additionally, kernel modules which are dynamically inserted into a kernel at run-time, run with the same privilege as the kernel proper. This means that if the modules are left un-measured, an attacker could use a malicious kernel module to perform an attack. Similar to the kernel proper, each module has its own code and read-only data sections, which means that we can measure those sections to attest the correctness of the module.

When dealing with the modules an additional complication arises and that is that kernel modules must be relocatable so they can be flexibly inserted into the kernel's address space alongside other kernel modules. Due to this fact, kernel modules must be compiled in a position independent manner. This means that branch/jump targets and references to external symbols must be left in a state that allows them to be adjusted when they are loaded (insertion time for kernel modules). The challenge here is that ideally we would like to measure the module's code and read-only data sections, including branch/jump targets and external references, in such a way that the measurement is reproducible on the verifier machine.

To solve this problem we design and make use of a VM profiler that generates a signature of a given VM's kernel and modules that is used during run-time to attest the correctness of the VM's OS. The following subsections explain in detail how the VM profiler works.

3.1 Obtaining VM Data

The profiler can work in an online mode or an offline mode. In either case it needs the state of the VM while it is running with all modules it might encounter during normal operation loaded. In online mode the profiler uses the name of the VM to request pages of memory from the VM via the Xen hypervisor. In offline mode the profiler uses a core-dump of the VM obtained by the user to obtain information about the VM. This is the safer mode of operation since it avoids any potential race conditions. Since it cannot be core-dumped, Domain-0 can only be profiled in online mode.

3.2 Profiling the Static Sections of the Kernel

The profiler first needs to profile the static portions of the VMs kernel. That is the code and read-only data of the kernel. Since the Linux kernel can load itself to different addresses and the kernel can vary in location and size of the code and read-only data sections, the profiler needs information about the layout of the kernel. This is obtained via the `System.map` file produced during the compilation of the kernel. Now knowing the start and end of the code and read-only data sections of the kernel, the profiler can request the appropriate pages of memory from the running VM (or core file), make adjustments for SMP locks and the jump table (see Section 3.3.3) and produce a SHA-1 hash of these sections. As an optimization for signature matching, a SHA-1 hash is also produced of the first page of the read-only data section of the kernel to act as a thumb print to select potential signature candidates during attestation. This first page happens to contain version numbers and build information about the kernel making it very likely to be a distinguishing characteristic of the VM. Our choice of the SHA-1 hashing algorithm is explained further in Section 5.

3.3 Profiling the Modules

Once the static portions of the kernel have been profiled, the profiler moves on to generating a profile of the kernel modules. In order to access modules and module information the profiler needs additional information about the kernel layout. In particular it needs to obtain the head pointer of the module list whose location is specified in the the `System.map` file. In addition to this pointer, the layout of the module structure is needed. Unfortunately this is not maintained in any distributed file and must be obtained from the kernel source. For most kernels a generic source can be downloaded from kernel.org since the module structure rarely changes. Once found the source will need to be configured in the same way as the loaded kernel. The profiler will automatically perform this step if the kernel configuration file is provided. Since parsing kernel source can be quite complicated and the kernel build system is already designed to do this, the profiler compiles a tiny, simple program called `dump_locs` against the kernel source to get the required information.

Once the the module list and module information is obtained a further complication exists. Namely, each time a VM boots it may load modules at different locations and in a different order. Given this need, modules must use relocatable code. In the module file is a relocation table that tells the kernel module loader what parts of the static kernel, other modules and even the module itself, to which the module will need references. Further, these references may be absolute or relative and 32-bits or 64-bits. Absolute references to itself or other modules and relative references to the kernel or other modules may change on each boot. Therefore, the profiler needs information about these relocations to properly profile the modules. This means the profiler must have access to the module files from the VM's drive.

3.3.1 Understanding Relocatable Code and the Relocation Table

The relocation table consists of entries of the following format:

Section	Location	Type	Symbol Reference

The section field is the section of the module where the relocation needs to go (i.e. "`.text`", "`.rodata`", etc.). The location field is the offset within the section where the relocation should be placed. The type field is the type of relocation to perform and is one of: 32-bit signed absolute reference, 64-bit absolute reference, 32-bit relative reference or 64-bit relative reference. Finally the symbol reference field is a description of the symbol to which the relocation should reference specified as a symbol name and offset (such as `printk-0x00000004`).

Each symbol exported by the kernel and other modules is maintained in a list in the kernel and has a unique name. Both exported and local symbols in the module itself are listed in the module file. The Linux module loader uses these lists to calculate the correct reference type to the requested symbol and place the reference in the module at the desired location.

Table 1: Relocation Offsets

	Kernel	Self	Other Module
Relative	+self.base_addr	no adjustment	+self.base_addr - mod.base_addr
Absolute	no adjustment	-self.base_addr	-mod.base_addr

3.3.2 Dealing with Relocations

The issue with relocations is that with each reboot of the VM a simple hash of the module's code and read-only data sections would change. To overcome this there are a couple of options. One would be to simply zero out all the relocatable references. Although fairly straight forward, this presents a potential security hole as these relocations are often calls to code in other modules and the static kernel. By zeroing out the relocations, an attacker could modify a module to reference its own code and the VM attestation code would not be able to detect this change. Therefore, we use a more complex solution of adjusting all relocations in the module in such a way that it would appear that the module and any module it references were loaded at address 0. In this way the information about which module was being referenced and the offset of the reference is preserved.

Rather than attempting to look up relocations we take advantage of the fact that the module loader has already done this for us. We know the base address of each module (from the module list parsing) and we know where the kernel starts. To determine what a relocation maps to we simply find the relocation in the VMs loaded module code, adjust for the relative position in the case of relative relocations, and determine whose memory space the relocation points to. We store which module (or kernel) this relocation points to then use Table 1 to adjust it.

As you can see, some relocations require no adjustment. Therefore, as an optimization we delete these relocations from our list so they need not be considered at attestation time.

The information stored to the profile has the following format:

Location	Size	Type	Reference Number

Location is the offset into the module's memory at which the relocation is performed. Size indicates how many bytes the relocation spans (typically 4 for 32-bit or 8 for 64-bit, but other values are possible as indicated in the next section. Type indicates whether this is absolute or relative (or a special case of zeroing out as described in Section 3.3.3). Reference Number is a number indicating which module the relocation points to (possibly itself or a special value for the kernel).

It may seem that an attacker could exploit this method by pointing the relocation to the same offset in a different module (most likely its own) since the base address of the referenced module is removed. However, since the adjustment is made both at profiling time and attestation time and at attestation time the base address removed is based on the module reference number, pointing to a different module will still give a different result and cause the attack to be detected. Furthermore, the addition of an unprofiled module will raise a flag in the attestation code and cause attestation to fail. Therefore even though we make it appear as though all modules were loaded at address 0 to get a consistent hash, attestation is still just as rigorous as it would be if we left the relocations alone.

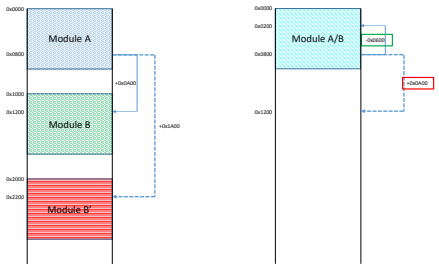

Figure 2: Failed Exploitation of Module Relocation.

As a more concrete example see Figure 2. In this figure module A is loaded at address 0x0000 and has a reference at offset 0x0800 to code at offset 0x0200 in module B which is loaded at address 0x1000. An attacker has loaded module B' to address 0x2000 which has attack code located at offset 0x0200. The attacker hopes that by removing the base address his new relocation will look identical to the valid one. However, since the attestation code removes the base address of module B, while the valid result should be -0x0600, the attacked relocation gives a result of +0x0A00 and the hash of module A will be incorrect.

3.3.3 Other Issues with Relocations and Solutions

One final set of issues comes with adjusting relocations on loaded modules. One is that the referenced code or data may have been unloaded already. Another is that the relocation may have been overwritten by code patching that occurred during module loading and further relocation may have moved with a patch.

Relative Text Relocations: Relative relocations

are always relative to the address of the relocation. However, in the x86 architecture relative jumps and calls are relative to the next program counter address. For this reason almost all relative relocations in the text section of a module are of the form `symbol-4`. An edge case exists when the relocation is to the first possible address in the target in which case the profiler would, without special consideration, incorrectly identify which module is being pointed to and identify the previous memory space instead. To overcome this we add 4 to all text relocations. This may seem to create a different edge case, but in practice the original problem is observed often, while the new edge case does not appear to happen.

References to Unloaded Sections: Once a module has finished loading and has been initialized, the kernel unloads all `.init.*` sections. As a result any relocations to this code or data will be broken and cause the profiler to misidentify the relocation. Since these relocations are not used after module initialization (this must be true or the module would be using a broken pointer), we simply look at the symbol name in the relocation table and delete (zero out) any relocation to a symbol starting with .init.

Alternate Instructions: To allow flexibility of the kernel and modules to be loaded on a number of different x86 processors but at the same time allow that same code to take advantage of advanced processor features, the Linux kernel uses a system of alternative instruction patching. That is, the main code section contains the most generic implementation that is guaranteed to work on all x86-64 processors, then in a separate section more processor specific code to replace that code with exists. These replacements along with a table specifying what processor features to look for to do the replacements allows the kernel to patch a module at load-time to be as efficient as possible. This isn't a problem for the static kernel since the replacements are always the same for the same hardware. However, for modules there may have been relocations in the replaced code and in the replacement code. Again, rather than try to identify the processor features and do the patching ourselves, we take advantage of the fact that the kernel has already done this. We take each piece of potential replacement code in the module file and adjust the references for the possible end location of this replacement. Then we compare the replacement code to the code found in the loaded module's text section. If it matches, then the replacement must have been performed and we should delete any relocations in the original code range and remap any relocations from the replacement code into this range.

Paravirtualized Instructions: Similar to the alternative instruction patching more recent Linux kernels contain native bare-metal hardware code in the main text section and can patch that code with calls to a hypervisor if paravirtualization is being utilized. However, unlike the alternative instructions where the replacement is packaged with the module. The paravitualized operation (PV-op) patches are packaged in the kernel itself and the module only indicates which PV-op should go where. Due to the complexity of this system we check to see if the original code is present and if not mark the code for zeroization at attestation time.

SMP Locks: On multi-core processors the `lock` opcode prefix causes a performance penalty even if only a single core is currently running. Therefore, as an optimization the Linux kernel and its modules specify a table of where these lock prefixes exist in the code. If the system is in single processor/core mode, then these lock prefixes are replaced with no-op instructions. This is especially concerning to our environment as VMs may be started with a variable number of VCPUs. To deal with this, we simply verify that the bytes whose addresses are listed in the SMP locks table are either a lock prefix (0xf0) or a no-op instruction (0x90) and then zero out the byte.

Jump Table: Originally intented to optimize the removal of debug code when the kernel is not in debug mode, the kernel jump table specifies the locations of conditional branch points where the condition variable changes very infrequently. Rather than using a conditional branch instruction in the code, the compiler writes an unconditional jump or no-op instructions based on the initial value of the condition variable. The condition variable is changed using a function call which identifies the affected branch points and rewrites the instruction(s). We deal with this in a way similar to SMP locks. We first verify that the targeted code is either a valid no-op sequence or a jump to the correct location. Once verified it is zeroed out to ensure a consistent hash result.

4 RUN-TIME ATTESTATION

In this section we provide details of how the VM profiles described in the previous section are incorporated into the run-time attestation protocol. The basic idea of the attestation protocol is similar to that of the Pioneer (Seshadri et al., 2005) and XSWAT (Ghosh et al., 2014) systems. The attestation protocol is shown in Figure 3 and it works as follows.

First the VM profiles are sent to both the verifier and the cloud machine. On the verifier machine a program is used to convert each individual VM's profile

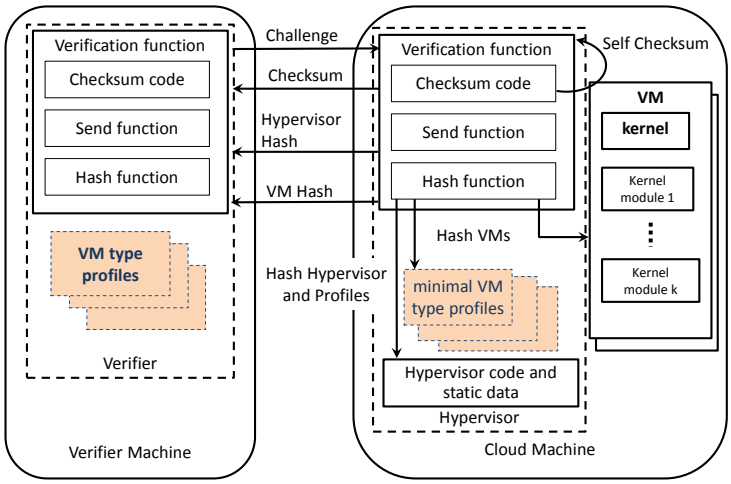

Figure 3: RIC Attestation Protocol.

into a signature after which the signatures of all the VMs (including Dom-0) that will run on the cloud machine are stored in a file. On the cloud machine each profile undergoes this conversion to a signature as part of the process of loading it into the memory space of the Xen hypervisor. In addition to the VM profiles, in order for the verifier to compute a known correct hash of the hypervisor the verifier requires the `xen-syms` file that is generated during compilation of Xen.

Attestations are initiated by the trusted verifier machine shown on the left side of Figure 3 who sends a challenge to the cloud machine which is to be attested. This challenge consists of a random nonce to be incorporated into the self-checksum performed by the cloud machine which is described below. The verification program that is run on the verifier machine is supplied with a *maximum* wait time and chooses a uniform random value between one and this specified maximum number of seconds to wait between attestation requests. This process assures than attacker cannot predict when attestations will be performed thus making it difficult to hide evidence of an attack beforehand.

The cloud machine begins by first disabling all maskable interrupts as well as replacing the non-maskable interrupt (NMI) handler with a handler that contains only a return instruction. This is done to ensure that an attacker has no means to hijack one of the cores during attestations and essentially leaves the attestation code in complete control of the cloud machine for the duration of the attestation. Next, the cloud machine performs a self-checksum over the verification function which contains the critical code necessary to carry out the rest of the attestation. In addition to the verificaiton function other critical pieces are also included in the checksum, including the newly replaced NMI handler. This ensures that if an attacker attempts to place the attack code in a NMI handler and later trigger an NMI, this change will be detected by the resulting checksum mismatch. The verification function includes the checksum code itself, the send function used to communicate with the verifier, and the hash function that is used to measure the rest of the software. Furthermore the challenge that was sent to the cloud machine from the verifier is also included in the checksum to eliminate the possibility of a replay attacks. The self-checksum provides a root of trust in software on the cloud machine which is extended further upward through the software stack by the rest of the attestation process. Once the checksum has been completed by the cloud machine the result is sent to the verifier using the send function. The verifier having the challenge and a correct copy of the verification function computes the same checksum to compare against the result it receives from the cloud machine.

The final portion of the attestation protocol is performing the hashes that attest the correctness of the hypervisor and guest OSes. Since the hashes do not change for a specific Xen hypervisor binary and the corresponding VMs that will run on top of it, the verifier's `verify` program can compute all of the necessary hashes once during initialization and store them for later comparisons. The cloud machine on the other hand must compute a set of fresh hashes during each attestation. The first item to be hashed is the hypervisor and includes its code section, read-only data section and additionally the VM signatures which reside in Xen's address space. Once this hash is finished, it is sent via the send function to the verifier. Depending on the performance budget allocated to attestation and the number of running VMs either all of the VMs or a randomly selected subset of them are to

be hashed next. For each selected VM, the guest's OS kernel code and read-only data sections are hashed. Then for each module that is currently running inside that guest VM, the relocation adjustment described in the previous section is performed and then a hash is computed over the module's memory space. It should be noted that these relocations and the resulting hashing are performed on a fresh copy of the VM module content to avoid having to undo all of the modifications once the hash is finished. Once the attestation request has been completed the cloud machine cleans up by restoring the normal NMI handler and re-enabling maskable interrupts. Finally, the trusted verifier can compare the checksum and hashes that it received from the cloud machine to the copies that it has computed locally to detect any mismatches and thus report them to the cloud owners.

5 PARALLELIZING HASH COMPUTATIONS

In this section we provide the details of our work in speeding up the RIC attestation process by parallelizing the hash computations that are utilized. Due to the fact that the guest OS is now hashed in addition to the Xen hypervisor the amount of time spent hashing during each attestation request represents the majority of the time taken to complete each attestation. The original XSWAT system doesn't allow an attacker to utilize extra processors by halting all but the bootstrap processor (which performs all the computations sequentially) during requests so all but one of the processors in a multi-core system are unused. Making use of these otherwise idle processors comes with no loss of security so long as the attacker is not allowed to use those processors to hide evidence of an attack during attestation, which is guaranteed by the attestation protocol as discussed in Section 4.

5.1 Chunk-based Hashing

We call the general idea of our hashing scheme *chunk-based hashing*. Our prototype uses the SHA-1 hashing algorithm, which is inherently a sequential hashing algorithm due to its "chained" nature. The output of each hash block is used as an input to the hash of the next block. This sequential process is depicted at the top of Figure 4. In this work we chose the SHA-1 hashing algorithm due to export control restrictions. However chunk-based hashing can be adapted to use other hash functions such as SHA-2 or SHA-3 which are considered to be more secure. The specific choice of a hashing algorithm is not central to the ideas presented in this work and a cloud provider wishing to implement this system should choose a hashing algorithm whose strength (i.e. its collision-resistance property) fits their needs.

Chunk-based hashing in contrast to the chained serial hashing works as follows. Each block of the input data is treated as a separate block of data and hashed individually. Assuming that the input data contains N blocks (64 bytes each for SHA-1) this produces N output hashes (20 bytes each). These output hashes, which we refer to as intermediate hashes, are then concatenated and treated as the input to the next level of hashing. Similarly to the top level, the intermediate hashes are divided into M blocks (with $M < N$) and again each is hashed individually producing M output hashes. This process continues until a level consists of a single block of input data and thus results in a single hash. An example of this process that shows a single level of this hashing scheme is depicted at the bottom of Figure 4. This hashing scheme readily allows parallelization of the hash computations since a separate hash is computed for each input block, which does not depend on any other block(s).

Assuming the use of a cryptographically secure hash function this construct is no less secure than hashing sequentially. In this model the attacker only has the ability to modify the initial input data and is not able to directly manipulate the intermediate hash results. Intuitively if the attacker wants to cause a collision in the final hash, which we will call the N^{th} level hash, they can cause a collision in any of the $N-1$ levels below the N^{th} level hash. In theory this presents more opportunity for the attacker to cause a collision in the final hash, however as previously mentioned the attacker can only modify the initial input data. This means that the attacker needs to make some modification to the initial input data that results in either an immediate collision in the first level of hashes or an output value that will cause a collision in the next level(s) of hashes. Given a cryptographically secure hashing algorithm which includes a collision resistance property, finding even an immediate collision in the first level of hashing is difficult. Furthermore since the attacker only has indirect control over the inputs to the rest of the levels of the hash causing a collision in these levels is also quite difficult. For a more rigorous proof of the security of this approach see Appendix 1.

5.2 Coordinating Hashes

The chunk-based hashing design presented in the previous subsection is coordinated using a lock-less work

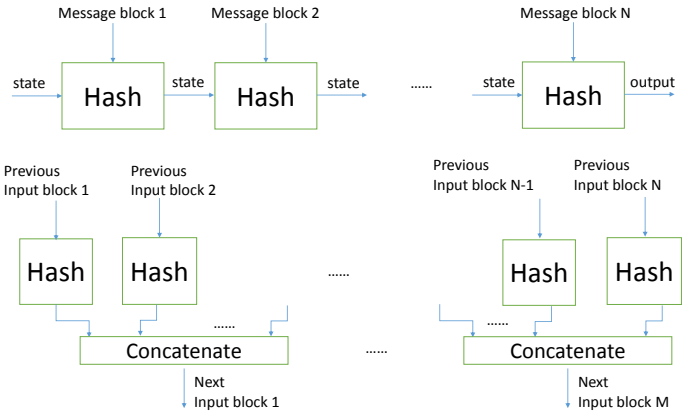

Figure 4: Sequential vs Parallel Hashing.

queue. The work queue is managed by a single master thread and hashes are carried out by a number of worker threads that watch the queue for items to hash, hash them, and write the resulting hashes into a buffer. The work queue consists of two arrays, one to hold work queue items and one to hold the results of the intermediate hashes. Each work queue item represents a single hash to be performed and includes all of the necessary data required to perform the hash such as the location of the input data, where to put the hash output, the size of the input etc. The necessary capacity of the work queue can be computed during its initialization based on the maximum amount of data that it needs to be able to process. This means that the arrays can be allocated once during initialization time rather than at the beginning of each hash calculation.

It is necessary to keep track of the number of workers currently working on each item to avoid use-after-free errors that arise from freeing buffers that hold input data while hashes of that data are being computed. Our implementation uses the atomic increment and decrement instructions offered by the x86-64 architecture to update the worker count safely across multiple processors.

After the master thread sets up each work queue item they can be in one of the following three states: The `UNCLAIMED` state which means that it is ready to be picked up and handled by one or more workers. When a worker picks up an item the worker changes its state to `CLAIMED` (and increments the worker count). Finally when a worker completes the hash denoted by an item it transitions to the `COMPLETE` state to signal that the corresponding result is ready.

To avoid all of the workers repeatedly picking up the same queue item to hash, workers start their search for an item to hash based on the processor ID of the processor on which they execute. If the worker checks all of the items and does not find one in the `UNCLAIMED` state and it has been configured to do so (at compile time) it begins its search again this time searching for items in the `CLAIMED` state. This offers some possible redundancy in the computation of each item's hash. There are number of reasons why redundancy might provide an advantage. The first would be that it offers fault tolerance. If for any reason a worker is unable to complete the hash it will be completed by another worker. This may also provide a performance boost in the presence of heterogeneous cores, for example ARM's big.LITTLE technology (Greenhalgh, 2011) or Dynamic Voltage and Frequency Scaling (DVFS) (Semeraro et al., 2002; Le Sueur and Heiser, 2010) where a slower core may pick up an item first but a faster core that becomes available later may be able to finish it faster. Due to the lock-less design of the work queue, even though it is not explicitly allowed, more than one worker can process the same hash item due to race conditions in setting and reading a queue item's state. While this does happen it is rather infrequent. It should be noted that the absence of locking does not have an effect on the correctness of the output and corresponding change of state to `COMPLETE` when an item is finished since the result being written and the state change are identical for any workers who happen to write them simultaneously.

5.3 Improvements and Optimizations

The first improvement to the above design addresses two issues and this improvement is where the name chunk-based hashing comes from. The first issue is the amount of extra data that needs to be hashed due to this scheme. This corresponds directly to the number of intermediate hashes. For a 64MB file the increase in the amount of data that needs to be hashed is roughly 1.5X for a total of 100MB (64MB of input

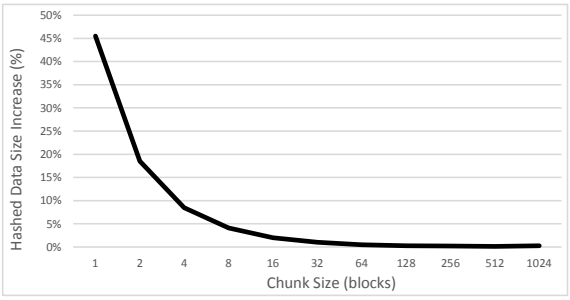

Figure 5: Data Size Increase vs Chunk Size.

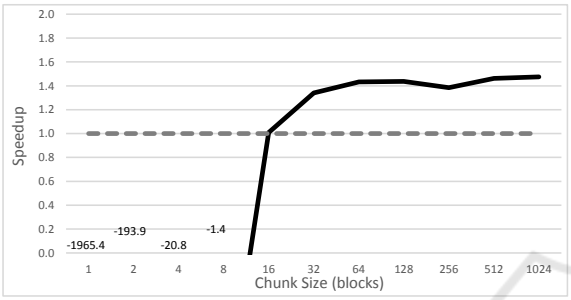

Figure 6: Speedup vs Chunk Size.

data and 32MB of intermediate hashes). In addition to the fact that more data needs to be hashed overall, each call to the SHA-1 hashing function comes with various setup and tear down overheads. If this setup and tear down cost is incurred for each individual block of input data and each block of intermediate hashes the overall performance degrades quite significantly. Indeed if each queue item handles only a single SHA-1 block a performance loss of about 2000X is observed for a 64MB file when compared to the standard sequential version.

Our solution to this problem is, rather than each queue item handling a single 64 byte block, they instead are each assigned a chunk consisting of multiple blocks. Figures 5 and 6 show the reduction in added data and speedup attained compared to the sequential hashing respectively as the chunk size is increased. The optimal number of blocks in each chunk is a function of the size of the input data and the number of processors available. To this end the work queue dynamically determines the optimal chunk size at each level given these two values using Equation 1. The size of each chunk is then computed using this value and the size of the input data.

$$chunks = \frac{data\_size}{2 \times NUM\_CPUs} \quad (1)$$

The second improvement arises from the observation that if the master thread only performs queue operations, it spends a significant amount of time waiting for the workers to finish their work before it can insert more items into the queue or declare the hash complete. Addressing this problem is relatively simple and consists of allowing the master thread to help computing hashes between queue management operations.

The final optimization that we implemented is similar in nature to an optimization commonly made to the quick sort sorting algorithm. It is common in quick sort implementations to use a different sorting algorithm (such as insertion sort) when the partitions become smaller than a certain size. In this case, when the input data is sufficiently small, the regular sequential hashing technique out performs the parallel version. We experimentally determined the point at which the parallel version actually out performs the regular implementation for the specific testing machine that we used to be 16KB of input data. This information is used in two cases: 1.) if the initial data passed to our algorithm is less than 16KB the work queue isn't used at all and the data is simply hashed sequentially and 2.) when the intermediate hashes at a given level are smaller than 16KB the master thread signals to the workers that the hash is finished and computes the final hash sequentially.

6 ETHERNET DRIVER

XSWAT was devised to perform attestation over a serial link due to a lack of Ethernet support in Xen. While use of serial communication links may be infeasible on a large scale, most cloud operator environments make use of Ethernet based management networks that could be leveraged for attestation. Towards this end, a specialized PCI Ethernet driver was developed for the Xen hypervisor and incorporated into RIC. The driver handles hardware interrupts from the Ethernet card, and, on the upstream path, either identifies a frame as a RIC frame and handles it, or passes the frame to the VM (usually Xen's Domain 0) containing the appropriate Ethernet drivers. No downstream interference from the VMs is expected since all VMs are stopped when attestation is in progress. Linux code for the Ethernet driver was adapted for this purpose.

An incoming attestation request generates an interrupt that is caught by the RIC driver which confirms that the source of the interrupt is in fact the Network Interface Card (NIC). If it is further determined that the interrupt signaled a receive (RX) event, Xen's Domain 0 VM is paused. The NIC registers are read for the address of the packet in Domain 0 memory, the packet is read from this address and is matched against the byte signature for an attestation request.

In case there is no match, Domain 0 is unpaused otherwise the attestation code is triggered.

Once the attestation request processing is completed at the cloud machine, RIC then initiates the process of creating an outbound attestation response. Attestation response data is inserted into an outbound packet which needs to be inserted into DMA accessible memory. Since it is hard to allocate DMA memory from Xen, an existing packet in DMA memory is saved along with the state of the Transmission (TX) ring buffer and is then overwritten with the outbound attestation response. Once the response packet has been transmitted using DMA, the saved packet is restored, the TX ring buffer is restored to its previous state and Domain 0 is unpaused.

Both inbound attestation request processing and outbound attestation response processing are designed to have low overheads to reduce the possibility of introducing any jitter to the checksum process. The evaluation results presented in Section 7 show that no significant jitter is introduced by the RIC Ethernet driver.

7 EXPERIMENTAL RESULTS

In this section we present the results of our experiments in which we quantify the overhead introduced by hashing VMs in addition to the Xen hypervisor and the savings provided by utilizing the parallel hashing technique to perform these hashes. The machines used to capture our performance results are Dell R620 Xeon systems with 12 cores. Each core supports two hyperthreads resulting in a total of 24 total hyperthreads. The cloud machine and the verifier are connected using ethernet via two switches, one for management tasks (including the attestation traffic) and one for non-management tasks. The testbed setup is shown in Figure 7. A serial link between the verifier and the cloud machine was also used to run attestation transactions for the purpose of comparison against attestation performance over Ethernet.

7.1 Parallel Hashing Results

To measure the performance gained by using the parallel hashing scheme without any of the additional overheads from performing relocations etc. we ran a user-space implementation of the parallel hashing algorithm. The user-space implementation reads a file filled with random data into memory, launches the workers as pthreads (in the parallel case), and then measures the time that it takes to hash the contents of the file. For these tests we utilized all of the logical

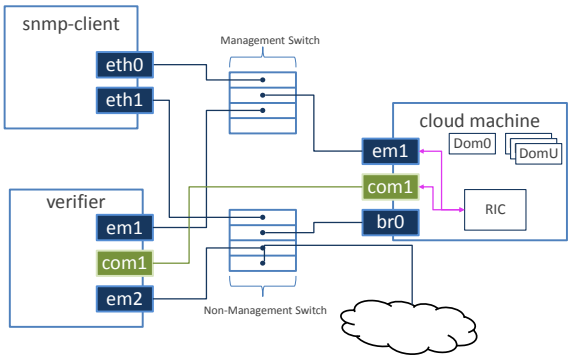

Figure 7: RIC Testbed Setup.

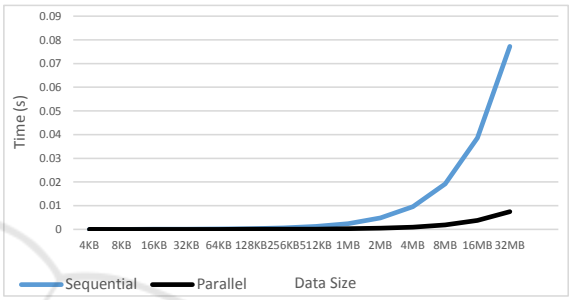

Figure 8: Parallel vs Sequential: Timing for Various Data Sizes.

Figure 9: Parallel vs Sequential: Speedup.

processors (hyperthreads) by using 23 worker threads and the single master thread. These measurements do not include the initialization of the work queue since that action takes place only once during boot of the Xen hypervisor and is reused from one hash to another. The work queue however does need to be reset after each hash and this overhead is included in the timings presented. Figure 8 shows the amount of time it takes the sequential and parallel hashing schemes to complete a hash of the given size for increasing data sizes. Figure 9 shows the speedup achieved by the parallel version when compared to the sequential version. Each data point in both graphs represents the average of 100 tests. As can be seen from the graphs more data being hashed results in a larger speedup for the parallel version which levels off at

around the 10X speedup mark. This 10X speedup mark is first attained with hashes around the 1-2MB mark in size. We believe that the reason that the maximum speedup achieved is not closer to 24X (the number of hyperthreads) is that the hyperthreads are doing identical work and thus there is a large amount of contention for micro-architectural resources that are shared amongst the hyperthreads that share the same physical core. To confirm this we disabled hyperthreading in the BIOS options and reran the experiments. The results were very similar to the results with hyperthreading enabled.

We also analyzed how using the parallel hashing technique affects the RIC attestation process as a whole. Particularly we were interested in whether or not the conditions of the hashes in the attestation protocol are suitable to achieve the 10X speedup that we observed for the larger data sizes above.

As can be seen in Table 2 the checksum time was not affected by the changes to the hashing algorithm as expected. The hypervisor hash experiences an optimal speedup of about 10X. The VM hash does not attain quite as much speedup as the hypervisor for a few reasons. First, due to the way timings are taken, the VM hash times also include the work of performing relocations as described in Section 3. In addition to this the VM hashing is less ideal than the hypervisor hash in that the hypervisor hash consists of a small number of larger hashes, while the VM contains a larger number of hashes some of which are as small as 4KB.

7.2 Ethernet Attestation Results

The performance of attestation over Ethernet was compared against that of attestation over a serial communication link in the context of a memory copy attack. In a memory copy attack, an adversary maintains a copy of the unmodified Attestation System in memory while using a malicious version of the code to compute the self checksum. While servicing an attestation request, the malicious code uses the unmodified code to compute the correct self checksum for the attestation system while hiding any evidence of cloud software environment tampering. We refer the reader to (Seshadri et al., 2005) for additional details.

The goal of this set of experiments was to determine whether conducting attestation over Ethernet would impact (a) the jitter of the self checksum process and therefore (b) cause false negatives in the face of a memory copy attack. In each of the experiments described in this section, 100 attestation requests were successfully completed to compute each data point. The average interval between successive attestation requests was set to 1 second. The number of checksum loop iterations was varied between 1,024,000 and 2,048,000 and exactly one randomly selected VM was hashed during the attestation process. The set of candidate VMs for hashing included Domain 0 and benchmark applications were run on each VM other than the Domain 0 VM. Each VM was allocated 2GB RAM and assigned 1 CPU except for Domain 0 which was assigned 2 CPUs. A total of 11 VMs were run on the cloud machine.

Table 3 shows checksum and attestation times over Ethernet and Serial interfaces. In all cases, the time taken over Serial is somewhat higher than that over Ethernet due to the former being somewhat slower than the latter. In all cases the checksum jitter is comparable across the two interface types.

The table also shows the time taken to execute a memory copy attack for a given number of checksum loop iterations. There is a clear separation between the checksum time and the time taken to conduct a memory copy attack indicating that Ethernet based attestation is fairly robust against these attacks. Self checksum performance was also assessed in the presence of cross traffic representative of network management applications. This cross traffic was generated by using an SNMP v2 client running on the snmp-client machine shown in Figure 7 sending traffic to an SNMP server running in Domain 0. Even with this cross traffic, self checksum times remained stable providing further evidence of the robustness of Ethernet based attestation as supported by RIC.

8 RELATED WORK

The RIC system is similar to the Checkmate (Kovah et al., 2012), Pioneer (Seshadri et al., 2005), MT-SRoT (Yan et al., 2011), and HyperSentry (Azab et al., 2010) systems and represents a direct extension of the XSWAT (Ghosh et al., 2014) system. RIC, while similar in spirit to these works, offers important improvements to both the security and performance of these systems. While each of these works uses timing-based attestation techniques to create a root of trust in software, this is the first of these works to extend that root of trust beyond the most privileged software layer upwards towards the user-space applications. This has important implications that are of particular interest to cloud providers. Using this technology a cloud provider can be assured that none of the hypervisors or operating systems on which their cloud services ultimately run have been compromised.

The hash parallelization scheme presented in this paper is somewhat similar in nature to

Table 2: XSWAT Results.

Activity	Sequential Time (ms)	Portion of Request	Parallel Time (ms)	Portion of Request	Speedup
1024K Checksum Iterations	13.41	30.81%	13.16	72.55%	1.0
Xen Hyervisor Hash	4.32	9.92%	0.44	2.43%	9.8
VM Hash	25.80	59.27%	4.54	25.03%	5.7
Total	43.53		18.14		2.4

Table 3: Ethernet Self Checksum Results.

Self Checksum Type	Iterations	Mean Time (ms)	Std. Deviation
Ethernet	1024K	13.56	0.08
	2048K	25.56	0.09
Serial	1024K	13.85	0.08
	2048K	25.84	0.09
Memory Copy Attack	1024K	15.03	0.2
	2048K	29.03	0.23

Merkle trees (Merkle, 1982), hash lists, and hash chains (Lamport, 1981). The general principle behind all of these constructs is to, perhaps repeatedly, apply a cryptographic hash function first to the input data and then to the output(s) that are created from this step. Each construct aims to solve a different problem whether it be increasing security or efficiency. Most similar to our work is Merkle trees which were originally designed as a scheme to generate digital signatures, however they can be applied to other problems. In fact one of the most common uses of Merkle trees today is to authenticate large data structures or files in an efficient manner. For example Merkle trees are used in peer-to-peer networks to allow only part of a file (or group of files) which is being downloaded from a given peer to be authenticated without the need to have the entire file. Merkle trees help peer-to-peer networks to be more efficient since if a single piece of a file is corrupted, either due to transmission errors or an attempted forgery, the Merkle tree allows the client to determine the specific block of the file that was corrupted and download only that block again rather than re-downloading the entire file. Our solution on the other hand is not concerned with identifying which piece of the measured data has been corrupted and instead is focused on speeding up the computation of the large hashes that the system must perform. To accomplish this, our technique is designed to run in a cloud environment which very often runs on a multi-core platform and thus is designed to explicitly take advantage of the availability of these cores to compute hashes in a parallel fashion. Recently, some other hashing constructions have been proposed (Maurer and Tessaro, 2007; Haitner et al., 2015) which are more similar in spirit to chunk-based hashing and could potentially require less rounds of parallel hashes resulting in a faster hash computation. Due to the modular design of the RIC system, the chunk-based hashing algorithm could be replaced by one of these constructs to further improve its performance and this possibility is being investigated.

9 CONCLUSION

In this paper we have described a method to profile and attest the correctness of a guest VM's operating system. This includes attesting the integrity of jump and branch targets as well as relocations in the dynamically lodabale kernel modules. In addition, to combat the added overhead of hashing the VM's guest OS, we also propose a parallel hashing technique that can increase the performance of hashing large areas of memory by 10X on a 12-core machine.

REFERENCES

(2007). TPM Main Specification. Accessed online at: http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. (2010). Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM.

Damgård, I. B. (1990). A design principle for hash functions. In *Advances in Cryptology–CRYPTO 1989 Proceedings*, pages 416–427. Springer.

Ghosh, A., Sapello, A., Poylisher, A., Chiang, C. J., Kubota, A., and Matsunaka, T. (2014). On the Feasibility of Deploying Software Attestation in Cloud Environments. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 128–135. IEEE.

Greenhalgh, P. (2011). Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8.

Haitner, I., Ishai, Y., Omri, E., and Shaltiel, R. (2015). Parallel hashing via list recoverability. In *Advances in Cryptology–CRYPTO 2015*, pages 173–190. Springer.

Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., and Butterworth, J. (2012). New results for timing-based attestation. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 239–253. IEEE.

Lamport, L. (1981). Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772.

Le Sueur, E. and Heiser, G. (2010). Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association.

Maurer, U. and Tessaro, S. (2007). Domain extension of public random functions: Beyond the birthday barrier. In *Advances in Cryptology–CRYPTO 2007*, pages 187–204. Springer.

Merkle, R. C. (1982). Method of providing digital signatures. US Patent 4,309,569.

Merkle, R. C. (1990). A certified digital signature. In *Advances in Cryptology–CRYPTO 1989 Proceedings*, pages 218–238. Springer.

Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D. H., Dwarkadas, S., and Scott, M. L. (2002). Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40. IEEE.

Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. (2005). Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, 39(5):1–16.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561. ACM.

Symantec (2015). 2015 Internet Security Threat Report. Accessed online at: https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf.

Wojtczuk, R. (2008). Subverting the xen hypervisor. *Black Hat USA*, 2008.

Yan, Q., Han, J., Li, Y., Deng, R. H., and Li, T. (2011). A software-based root-of-trust primitive on multicore platforms. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 334–343. ACM.

[Appendix 1]

In this section we formally define the chunk-based hashing construction, and prove it cryptographically secure, under the assumption that so is the basic hash function used as a component. The notion of security we use is that of collision-resistance (also known as collision-intractability) as typically done in the cryptography literature.

Informally speaking, a function H is collision-resistant if it is hard for an efficient adversary, who is given the full code of H, to find two inputs x, x' such that $x \neq x'$ and $H(x) = H(x')$, unless with extremely small probability. Let $H : \{0,1\}^b \rightarrow \{0,1\}^L$ denote a collision-resistant hash function, for integers $b > L > 0$ (e.g., if H=SHA256, then L=256). Based on H, we define a chunk-based hash function *cbH* by repeated parallel applications of H to the current input sequence, until the latter is compressed to a single block. The current input sequence is first set to the input to the chunk-based hash function, and from then on as the output of the parallel applications of H, after concatenation and padding. As defined so far, this function can be proved to be collision-resistant if so is H and if the function is only evaluated on inputs of the same, predefined, length. To efficiently extend this function to one that remains collision-resistant even when it is evaluated on inputs of different lengths, function *cbH* pads the computed single block with the input length, and then uses this string as the input for one last hashing step using H. The computed block will be the output of hash function *cbH*. Padding the sequence of input blocks with the input length has been used for similar reasons in Merkle-Damgaard's fully-sequential paradigm for extension of the domain of collision-resistant hash functions (Merkle, 1990; Damgård, 1990).

Let N be an integer > 0, and consider the chunk-based hash function $cbH : \{0,1\}^{Nb} \rightarrow \{0,1\}^L$ defined as follows. On input a sequence of N blocks $x = x[1],...,x[N]$, each of length b, *cbH* works as showin in Algorithm 1.

Algorithm 1: Chunk-Based Hashing Algorithm.

1: Set $i = 1, N(1) = N$
2: **repeat**
3: Let $y[j] = H(x[j])$, for $j = 1,...,N(i)$
4: Concatenate $y[1],...,y[N(i)]$ into $M(i)$ blocks $t[1],...,t[M(i)]$, each of length b, possibly padding the last block to length b
5: Set $x[j] = t[j]$, for $j = 1,...,M(i)$
6: Set $i = i + 1$
7: Set $N(i) = M(i-1)$
8: **until** $M[i] = 1$
9: Set b-bit blocks $u = x[1]$ and $v = N * b$, and set $z = H(u|v)$
10: Output: z

We now claim and prove the main result for func-

tion cbH.

Theorem 1. *If H is collision-resistant then so is cbH*

To prove this theorem, we prove the equivalent contrapositive version: if cbH is not collision-resistant then so is H. This is done by showing that if an efficient adversary algorithm cbA finds a collision in cbH, then we can construct an efficient adversary algorithm A that finds a collision in H, with the same probability. Let (x,x') be the collision found by cbA in cbH, where $x = x[1],...,x[N]$ and $x' = x'[1],...,x'[N']$. We distinguish two cases: $N \neq N'$, and $N = N'$.

Case $N \neq N'$: Note that $z = H(x[1]|N*b)$ and $z = H(x'[1]|N'*b)$, where $x[1]$ and $x'[1]$ here denote the values computed at the end of step 8 of cbH on input, respectively, x and x'. Because $N \neq N'$, the pair $(x[1]|N*b, x'[1]|N'*b)$ is a collision for H.

Case $N = N'$. Note that $z = H(x[1]|N*b)$ and $z = H(x'[1]|N*b)$, where $x[1]$ and $x'[1]$ are the values computed at the end of step 8 of cbH on input, respectively, x and x'. Let $i(max)$ denote the highest index such that $x[j] \neq x'[j]$, for $j = 1,...,M(i(max))$, where $x[1],...,x[M(i(max))]$ and $x'[1],...,x'[M(i(max))]$ here denote the values computed at the $i(max)^{th}$ execution of the repeat loop in step 2 of cbH on input, respectively, x and x'. By the maximality of $i(max)$, we obtain that $H(x[j]) = H(x'[j])$, for at least one value j in $\{1,...,M(i(max))\}$, which implies that the value $(x[j],x'[j])$ satisfying this condition is a collision for H.