An In-VM Measuring Framework for Increasing Virtual Machine Security in Clouds

In this framework, a module measures executables running in virtual machines (VMs) and transfers the values to a trusted VM. Comparing those values to a reference table containing the trusted measurement values of running executables verifies the executable's status.

Qian Liu, Chuliang Weng, Minglu Li, and Yuan Luo
Shanghai Jiao Tong University

Cloud computing relies heavily on virtualization. Virtualization technology has developed rapidly because of the rapid decrease in hardware cost and concurrent increase in hardware computing power. A virtual machine monitor (VMM, also called a *hypervisor*) between the hardware and the OS enables multiple virtual machines (VMs) to run on top of a single physical machine. The VMM manages scheduling and dispatching the physical resources to the individual VMs as needed, and the VMs appear to users as separate computers. Widely used virtualization technologies include VMWare,[1] Xen,[2] Denali,[3] and the Kernel-Based Virtual Machine (KVM).

A particularly popular function that cloud computing provides is software as a service (SaaS), which lets users access, from any computer, applications they don't have to own. All customers share a single instance of the hosted application, with the virtualization system managing access. Because applications and computing resources are no longer fully under users' control, the biggest challenge for SaaS cloud computing systems is guaranteeing user-level security.

Because of the isolation of individual VMs supported by virtualization, many approaches to user-level security use a trusted VM to monitor guest VM status. (Guest VMs deliver services to customers.) This method is called *out-of-VM monitoring*,[4] because it's implemented outside the guest VMs. Because context switching between the guest VMs and the trusted VM creates a large performance overhead, out-of-VM monitoring isn't suitable for applications-intensive systems such as SaaS.

So, we developed a security-measuring framework suitable for SaaS. Our framework aims to determine the status of user-level applications in guest VMs that have run for a period of time. To do this, it combines measurement principles[5] with in-VM monitoring.[6] Unlike other in-VM monitoring approaches, our framework doesn't require processor virtualization technology. Because executables' contents shouldn't vary between measurements, our framework can detect attacks on user-level executables by noticing measurement changes. Moreover, to our knowledge, it will generate no false positives. In addition, it monitors the status of the measurement module in the guest VMs to guarantee that the measurement process is trusted. A prototype of this framework has demonstrated good performance efficiency.

Our Framework

In our framework (see Figure 1), a measurement module (MM) in each guest VM measures every running executable in that VM. The MM transfers new measurement values to the trusted VM via standard inter-VM communication mechanisms. The trusted VM stores those values in sequence in a measurement table (MT). At the same time, the system extends these measurement values into a specified platform configuration register (PCR). To ensure the measurement process's trustworthiness, we add a memory watcher (MW) module to the VMM.

We also measure the executables in a trusted environment—for example, without being connected to the Internet. The derived measurement values are stored in sequence in a reference table (RT) in the

trusted VM and are extended into another specified PCR. The PCRs aren't strictly necessary; we could verify an executable's status by comparing its measurement value in the MT with the corresponding trusted measurement value in the RT.

Assumptions

To simplify things, we assume that the OS kernel in the guest VM doesn't suffer any kernel-level attacks. To ensure that it doesn't, we could apply any of several approaches. (For more on these and other approaches related to VM security, see the sidebar.)

The open source Trusted Boot project (TBoot; http://sourceforge.net/projects/tboot) has focused on a software module that loads before the OS kernel or VMM and performs a measured and verified launch of those components. TBoot can guarantee that the VMM and the OS kernels in the VMs are trusted during booting. We assume that our framework boots with TBoot—that is, we assume our framework is operating in a trusted environment at the beginning.

Then, we assume that the VMM is a trusted node, because it has the highest privilege level. Likewise, we assume that our trusted VM is a trusted node, because it has a higher privilege than guest VMs. Because the MT and RT store sensitive data, we don't assume they're safe from attack even though they're both in the trusted VM. Instead, to ensure their integrity, we leverage the Trusted Platform Module (TPM).[7]

We also can't assume that the MM is safe from attack, because it's the crucial element of our in-VM measuring framework. So, we introduce the MW to protect the MM.

Our framework measures executables when they launch, which is effective for many attacks that affect or modify the contents of executables when they're not running. Dynamic attacks during runtime come in many different forms; so far, no security mechanism can resist all of them. So, our threat model doesn't account for dynamic attacks.

Finally, we make the standard assumptions of virtualization architectures that the trusted computing base includes the VMM and trusted VM but not the guest VMs. In other words, attacks on a guest VM can't affect the VMM and trusted VM.

Measurement

Our framework performs its measurements by computing a hash function over an executable's contents. Such a function has two innate properties. First, an input of arbitrary length generates a fixed-length output (the *hash value*). Second, different inputs can't generate the same outputs. This means that an executable's computed hash value can be its unique identifier.

Since the MM requires modifying the OS kernel, we selected Linux because it's open source. In Linux, every executable exists in a unified format in the virtual memory address space: the executable and linkable format (ELF). The MM computes a hash function for each executable's ELF contents when it launches (that is, before the executable is really running); the resulting hash value is the executable's measurement value. So, we can determine the executable's status by verifying its measurement values. (The same is also true for self-modifying codes because their contents are identical every time they start running.[5])

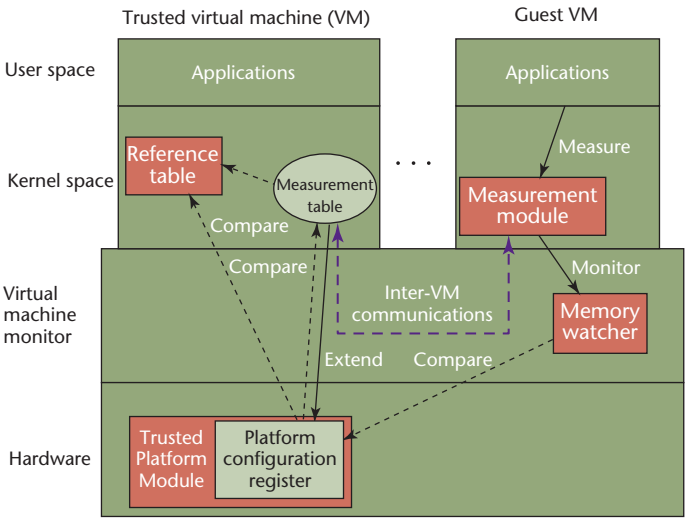

Figure 1. Our in-VM (virtual machine) measuring framework. This framework can effectively determine the status of user-level applications in the guest VM. Trusted Platform Module plays an important role in this framework.

Storing Measurement Values and Extending Them into a PCR

Once a value is stored in an MT in the trusted VM, we extend it into a PCR, a group of data registers inside TPM. To do this, we use the Linux PCR extension operation:

$$PCR\_Extend(PCR[i], m) = SHA1(PCR[i] \parallel m).$$

This operation is an SHA-1 computation. SHA-1 is a secure hash algorithm (SHA) that produces a 160-bit output from an input with a maximum length of $(2^{64} - 1)$ bits. Index i indicates which PCR is involved in the operation, the $\parallel$ operation represents concatenation of two inputs, and m is a measurement value.

We store this operation's hash value in the same PCR as that used for the computation, so $PCR[i]$ is both input and output in this operation. Consequently, the PCR value is the accumulative SHA-1 hash value of all generated measurement values. Two different PCRs hold the accumulative SHA-1 hash values of the measurements in the MT and RT. A third PCR holds the accumula-

Related Work in Virtual Machine Security

Researchers have proposed several approaches to address virtual machine (VM) security. The Integrity Measurement Architecture (IMA) aims to measure executables running in an OS and enable a third party to verify whether those executables are trusted.[1] This approach isn't specifically designed for virtualization architectures, however. Furthermore, IMA doesn't address the status of the measurement module in a guest VM, whereas our framework adds a memory watcher in a trusted VM to monitor the measurement module's status. So, our measuring framework offers higher security guarantees than IMA.

Attacks on OSs fall into two categories: kernel level and user level. One form of kernel-level attack is kernel-level rootkits, which could modify the hooks in kernel space to hide their presence. To prevent this, Zhi Wang and his colleagues proposed HookSafe, a lightweight hypervisor-based system.[2] HookSafe introduces a hook indirection layer that maps hooks to be protected into a continuous memory space. HookSafe then leverages hardware-based page-level protection to control access to this memory space.

Livewire is an intrusion detection system in the VM monitor that monitors the status of a VM's kernel.[3] It also sets the kernel code segments to be read-only to prevent malicious code injection.

SecVisor is a tiny hypervisor that uses hardware memory protection to protect kernel code integrity.[4]

Although most of these approaches to detecting kernel-level rootkits can detect attacks, they can't confirm what kind of attack it is. To solve this problem, John Levine and his colleagues proposed a framework that detects and classifies rootkits by comparing the difference between the original program and the rootkit-infected program.[5] This approach can distinguish between known kernel-level and new rootkits.

The Lares architecture, an example of out-of-VM monitoring, can actively monitor events in guest VMs.[6] It leverages virtualization's isolation capabilities to separate the VMs into a security VM and untrusted guest VMs. Lares inserts a hook into guest VMs that can invoke a security application in the security VM. That application in turn evaluates events in the guest VM.

Although the out-of-VM monitoring helps ensure security, frequent context switches between VMs introduces a high performance overhead that creates a bottleneck. So, Monirul Sharif and his colleagues proposed secure in-VM monitoring (SIM), a framework that places the security application into the untrusted VM.[7] To guarantee the same security level provided by out-of-VM monitoring, SIM uses hardware virtualization.

References

1. R. Sailer et al., "Design and Implementation of a TCG-Based Integrity Measurement Architecture," *Proc. 13th Usenix Security Symp.*, Usenix Assoc., 2004, p. 16.
2. Z. Wang et al., "Countering Kernel Rootkits with Lightweight Hook Protection," *Proc. 16th ACM Conf. Computer and Communications Security*, ACM Press, 2009, pp. 545–554.
3. T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," *Proc. Network and Distributed Systems Security Symp.* (NDSS 03), Internet Soc., 2003, pp. 253–285; http://suif.stanford.edu/papers/vmi-ndss03.pdf.
4. A. Seshadri et al., "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," *ACM SIGOPS Operating Systems Rev.*, vol. 41, no. 6, 2007, p. 350.
5. J. Levine, J. Grizzard, and H. Owen, "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security & Privacy*, vol. 4, no. 1, 2006, pp. 24–32.
6. B. Payne et al., "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *Proc. IEEE Symp. Security and Privacy*, IEEE Press, 2008, pp. 233–247.
7. M. Sharif et al., "Secure In-VM Monitoring Using Hardware Virtualization," *Proc. 16th ACM Conf. Computer and Communications Security*, ACM Press, 2009, pp. 477–487.

tive SHA-1 hash value of the trusted measurement values generated under a trusted environment.

We pair the measurement values in the MT and RT with the associated executables' names. During verification, the executables' names serve as indexes to the corresponding measurement values.

Verification

This process has two main parts: verifying the executables' status and verifying the MM's status.

Verifying executables' status. First, we need to ensure that the MT and RT are trusted. We use the same method to protect the two, so we'll look just at the MT.

Because the MT's PCR holds the accumulative SHA-1 hash value of the table's measurement values, we recompute the SHA-1 hash value for those values in sequence and compare that value with the value in the PCR. If the two values are equal, the MT is secure. After that, our framework compares the new measurement values in the MT to the corresponding values in the RT, indexed by the same executable's name. If the two corresponding values are equal, that executable is trusted. Otherwise, we assume an attack has occurred.

Verifying the MM's status. The MW measures the MM in each VM periodically. Each time, the MW recomputes the SHA-1 hash for the MM's measured values in sequence. If the result is equal to the trusted accumulative hash value in the relevant PCR, the MM is secure.

Implementation

Our prototype (see Figure 2) is based on the open

source Xen VMM. Xen is widely used in cloud computing—both Amazon's EC2 Service and Eucalyptus are based on it. It supports two running modes, full virtualization and paravirtualization. Full virtualization needs the support of processor virtualization technology, such as Intel's VT-x and AMD's Secure Virtual Machine. In the absence of such support, paravirtualization promises improved performance over full virtualization. Because we can't count on a given computer having hardware virtualization, our framework adopts paravirtualized Xen.

In the Xen context, a VM is a domain and the VMM is the hypervisor. The first domain to boot is Dom0, a management domain; it's equivalent to the trusted VM in our framework. Unprivileged domains, equivalent to our guest VMs, are DomU. The OS running in each domain is called a guest OS.

Measurement

The Linux Security Module offers a group of hook functions in `struct security_operations`. We implement measurement hook functions, with SHA-1 as the hash function, in the guest VM that are called before executables run. So, our implementation measures executables before they run.

Interdomain Communication

Xen implements several mechanisms for communication between domains, such as a grant table, ring buffers, an event channel, and XenStore. We use all those mechanisms to transfer the measurement values from DomU to Dom0.

We implement two modules for interdomain communication: a front-end module (FE) in DomU and a back-end module (BE) in Dom0 (see Figure 2). To transfer the measurement values, DomU first shares the ring buffers, which transfer measurement values by using a grant table. The key function in this process is `gnttab_grant_foreign_access()`. Subsequently, DomU establishes a new event channel for notifying Dom0 of a new measurement value, using the hypercall `HYPERVISOR_event_channel_op()`. XenStore stores DomU's grant reference and event channel port; Dom0 fetches this information by invoking `xenbus_scanf()`. Using a grant reference, Dom0 maps the shared ring buffers to its own memory address space by invoking `HYPERVISOR_grant_table_op()`. Using an event channel port, Dom0 invokes `bind_interdomain_evtchn_to_irqhandler()` to bind its own event channel to DomU and allocate a handler for events. This sequence of actions successfully transfers the measurement values from DomU to Dom0.

Protecting the MM and FE

We also take measurements of the MM and FE to determine their status. If we used mechanisms such as a grant table and shared memory to transfer the contents of the MM and FE from DomU to Dom0, we would need to implement an additional module in DomU. We can't guarantee this module's security, so we must find another method.

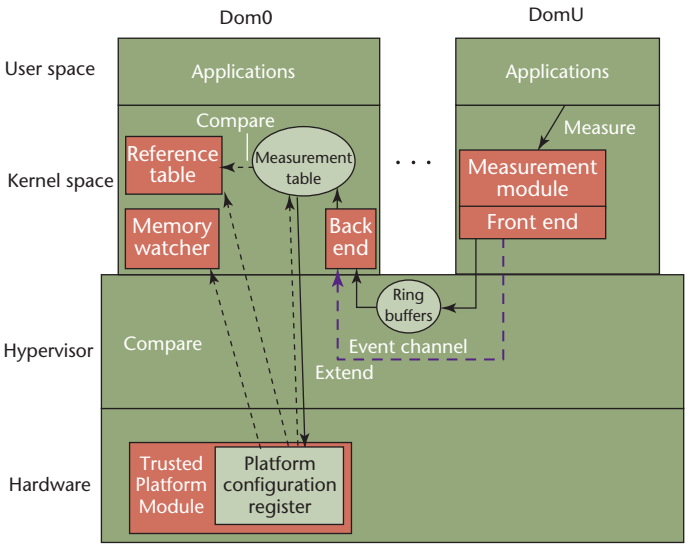

Figure 2. Our prototype implementation in paravirtualized Xen. Our in-VM measuring framework (shown in Figure 1) is implemented in paravirtualized Xen. Mechanisms provided by Xen, such as ring buffers and event channels, are utilized to transfer the measurement values from DomU to Dom0.

Instead, our framework leverages the Linux kernel file `System.map`. This file is a symbol table that the kernel uses to establish correspondence between the names of functions or variables and their virtual addresses in memory. Because the MM and FE are both modules in DomU's kernel, all the functions and variables they use are in DomU's `System.map` file, letting us retrieve their virtual addresses.

To do so, we leverage the fact that paravirtualized Xen uses three layers of memory: virtual, pseudophysical, and machine. The pseudophysical memory resides between the machine memory—the memory the hardware system actually uses—and guest domain OSs. It appears to each guest OS as its own machine memory. Xen assigns a machine frame number (MFN) and a physical frame number (PFN) to single pages of machine memory and pseudophysical memory, respectively. Each domain maintains a local table called P2M that converts a PFN to the corresponding MFN.

P2M is a two-level page table. In the shared info page, we find the MFN pointing to the first level of the page table in `domain -> shared info -> arch.pfn_to_mfn_frame_list_list` (`domain` is the domain to which this P2M belongs). Having

found the P2M table in this way, we next implement a function to find a given PFN's corresponding MFN. On the basis of this function, we add to Xen the hypercall `HYPERVISOR_map_by_vaddr_op`. This hypercall takes two arguments: the domain ID and the structure `map_request_t` containing operation parameters.

Specifically, we define `map_request_t` as

```
struct_map_request {
/ * IN parameters * /
domid_t domid;
unsigned long vaddr;
int size;
/ * OUT parameters * /
XEN_GUEST_HANDLE(ulong) start;
};
typedef struct map_ request map_
request_t;
```

With this hypercall, Dom0 can fetch the memory areas of DomU corresponding to virtual addresses `vaddr` to `vaddr + size`. Dom0 then accesses these fetched memory areas in its own virtual address space, from virtual addresses `start` to `start + size`.

Our prototype uses this technique to find DomU's `System.map` (`System.map-2.6.18.8-xenU`) in Dom0's directory `/boot`. Then, the MW maps all functions, read-only data, and initialized data used in the MM and FE to their virtual addresses in Dom0 by invoking `HYPERVISOR_map_by_vaddr_op`. The MW measures those virtual addresses according to their sequence in `System.map`. Under a trusted environment, the MW extends trusted measurement values into a specified PCR (an unused PCR) in sequence. The system then calculates the values for MM and FE during runtime. If the MM and FE are intact, two sets of measurement values should be the same.

Verification

The verification process in our Xen implementation is the same as we described in the section "Our Framework." We don't verify the executables' measurements periodically; rather, we verify them only when new measurement values are generated in DomU. At that time, we compare the new measurement values with the corresponding values in the RT.

In contrast, because the number of functions and data to be measured for verification of the MM and FE is invariable, we don't store the measurement values in any table. Instead, we take a measurement every five minutes and immediately compute the SHA-1 function for those values in sequence. If the result equals the PCR value holding the trusted accumulative hash value of all trusted measurement values, we conclude that the MM and FE are intact.

Security Analysis and Experiments

Here, we analyze our prototype's security and describe an experiment testing its ability to detect intrusions.

Security Analysis

This analysis involves four areas: transferring measurement values, PCR extension, measurement, and the MM and RT.

Transferring measurement values. The grant table guarantees that one domain's memory areas can be accessed only by another granted domain. In our prototype, DomU owns the shared ring buffers, and it grants only to Dom0 the rights to access these shared memory areas. So, the transfer of measurement values is secure.

PCR extension. In paravirtualized Xen, DomU accesses hardware indirectly. To guarantee the PCR extension's security, we transfer measurement values from DomU to Dom0 and perform the extension in Dom0. In Dom0, we directly access the PCR in the TPM, which guarantees the extension's security.

Measurement. As we mentioned before, by invoking `HYPERVISOR_map_by_vaddr_op`, the MW can map the MM and FE modules' contents. The MW computes these contents' hash values in sequence and performs an SHA-1 function on these measurement values in sequence. If the accumulative hash value equals the trusted accumulative hash value in the PCR, the MM and FE are secure.

The MT and RT. To confirm the MT's integrity, we first compute the SHA-1 function for every measurement value in it in sequence, as we mentioned before. We then compare the resulting hash value to the value in the corresponding PCR. If they're equal, the MT is intact. We use the same process to confirm the RT's integrity.

An Experiment

To test our prototype's ability to detect intrusions, we wrote a simple program in DomU, compiled the source code into an executable file, and ran it. Here's the program's main source code:

```
fd = open("hello.txt",O CREAT |O RDW R, \
          S IRUSR|S IWUSR);
if(fd){
  write(fd, "hello world!"
strlen("hello world!"));
  close(fd);
}
```

We named the executable `/home/piano/code/sys_file`; its corresponding measurement value is

0c772110d4f010ce3442f95e827e173789409bbd. Once we determined the measurement value, we modified the source code to change "hello world" to "HELLO WORLD":

```
fd = open("hello.txt",O CREAT |O RDW R, \
          S IRUSR|S IWUSR);
if(fd)
  {write(fd, "HELLO WORLD!", \
         strlen("HELLO W ORLD!"));
close(fd);
}
```

We gave the modified executable the same name. In this case, the corresponding measurement value in the MT is 2b0b56af6dbd91fc0ad2cfd035c451da7e202134. After comparing the two values to the corresponding trusted value in the RT, Dom0 discovered a mismatch. So, our prototype correctly detected the modification of the executable.

Performance Evaluation

Our measuring framework will incur additional, unavoidable computing overhead. In cloud computing, many applications run concurrently. Generally, the measuring framework would affect parallel programs more than sequential programs, so we tested parallel programs. We performed experiments with three benchmark suites from the Stanford Parallel Applications for Shared Memory:[8] Barnes, LU (contiguous blocks), and FFT (fast Fourier transform).

The Experiment Environment

We performed all the experiments on a Lenovo ThinkPad R61 with a dual-core Intel T8100 2.10-GHz CPU, 2 Gbytes of RAM, and on-chip TPM v1.2. We used Xen version 3.4.2. Our experiments involved domains Dom0 and Dom1. Each ran Ubuntu 8.10, with Linux kernel 2.6.18.8.

Experiment Parameters

Dom0 received two virtual CPUs and 1 Gbyte of memory, whereas Dom1 received one virtual CPU and 512 Mbytes of memory. Throughout the experiments, we kept Dom0's weight (its relative priority in access to CPU time) at 256, and we varied Dom1's weight from a minimum of 32 to a maximum of 512. We ran each benchmark in both domains, first without our prototype and then with it. For each experiment, the reported computation time is the average of 10 tests.

For Barnes, we set the fleaves parameter to 2.0 and the number of processors at 2; all other parameters were default values. For the LU contiguous-block-allocation test, n = 2,048, p = 2, and b = 16, where n is the order of a $N \times N$ matrix, p is the number of processors, and b is the block size. For the FFT, p = 2, n = 65,536, l = 4, and m = 22, where n is the number of cache lines, l is the logarithm (base 2) of the cache line length in bytes, and m is the exponent of 2. m can be any even integer; we chose 22 because it produced an appropriate execution time.

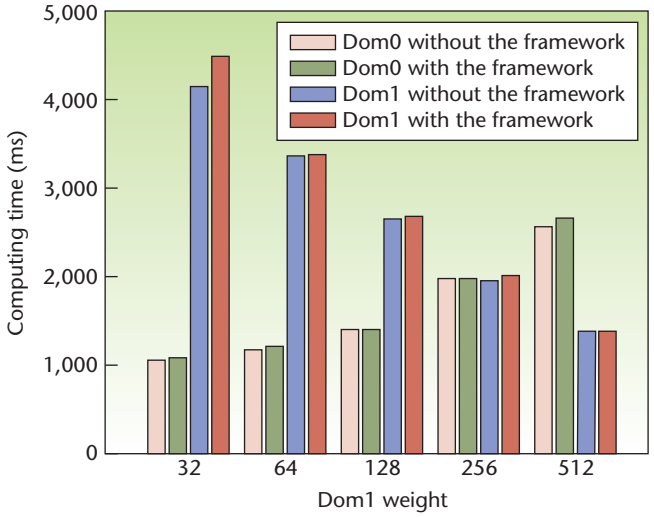

Figure 3. The Barnes workload. The Barnes application utilizes the Barnes-Hut algorithm, which simulates the interaction of a system of bodies (*N*-body problem) under the effect of the gravitational force.

Results

Figures 3 to 5 show the results. The computation times in Dom0 and Dom1 with our prototype were a few percentage points higher than those without (at most, 4.13 percent in Dom0 and 8.79 percent in Dom1). That indicates that our framework introduces low performance overhead.

In Dom1, the main overhead is due to measurement, with more added during transfer of measurement values. In Dom0, PCR extension causes some performance cost. During verification, the performance overhead is due to the recomputation of measurement values and their lookup in the reference table. Because these actions occur only when the system generates new measurement values, the overhead is modest.

Context switching between DomU and Dom0 occurs when DomU notifies Dom0 that new measurement values are available. It sends this notification only when the ring buffers are empty—that is, when DomU is waiting for new measurement values. If many measurement values are stacked in the ring buffers, DomU deals with them continuously. This constrains the overhead due to context switches.

Finally, invoking `HYPERVISOR_map_by_vadd_op` decreases efficiency. But in our prototype, the MW invokes this hypercall only every five minutes, so the overhead is acceptable.

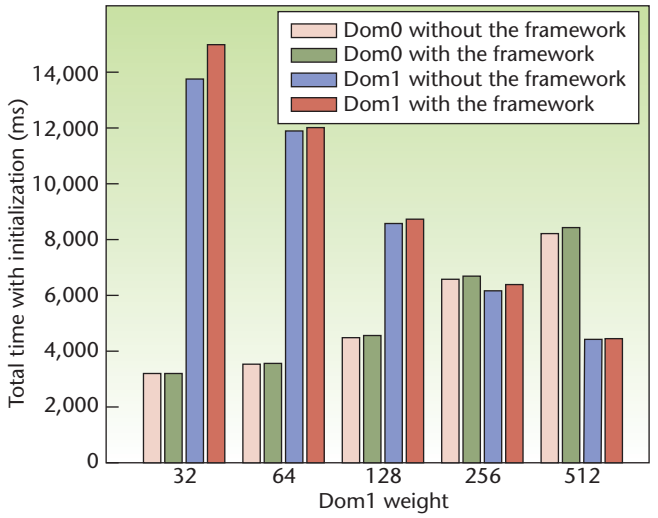

Figure 4. The LU workload. The LU kernel factors a dense matrix into the product of an upper and a lower triangular matrix. In order to exploit temporal locality on submatrix elements, the $n \times n$ dense matrix is decomposed into an $N \times N$ matrix of $B \times B$ blocks ($n = N \times B$).

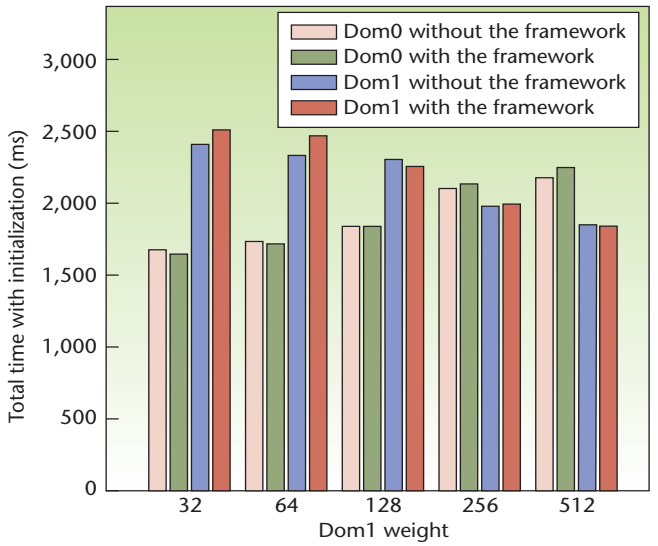

Figure 5. The FFT (fast Fourier transform) workload. The FFT kernel is based on the radix-$\sqrt{n}$ six-step FFT algorithm. In order to minimize interprocessor communication, the FFT kernel makes some optimization.

Our in-VM measuring framework is a practical way to determine the status of user-level applications in guest VMs. If the framework detects tampering, the VMM's rollback mechanism could restore guest VMs to their previous intact status. So far, this framework has only been implemented in a single physical machine. Because clouds involve multiple computers (servers) cooperating with each other, further research should focus on a measuring framework applicable to multiple physical machines. □

Acknowledgments

This work was supported by the National Key Basic Research and Development Plan of China (973 Plan, grant 2007CB310900) and the National Natural Science Foundation of China (grants 90612018, 90715030, and 60970008).

References

1. J. Sugerman, G. Venkitachalam, and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *Proc. Usenix Ann. Tech. Conf.*, Usenix Assoc., 2001, pp. 1–14.
2. P. Barham et al., "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles*, ACM Press, 2003, p. 177.
3. A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," *Proc. Usenix Ann. Tech. Conf.*, Usenix Assoc., 2002; http://denali.cs.washington.edu/pubs/distpubs/papers/denali_usenix2002.pdf.
4. B. Payne et al., "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *Proc. IEEE Symp. Security and Privacy*, IEEE Press, 2008, pp. 233–247.
5. R. Sailer et al., "Design and Implementation of a TCG-Based Integrity Measurement Architecture," *Proc. 13th Usenix Security Symp.*, Usenix Assoc., 2004, p. 16.
6. M. Sharif et al., "Secure In-VM Monitoring Using Hardware Virtualization," *Proc. 16th ACM Conf. Computer and Communications Security*, ACM Press, 2009, pp. 477–487.
7. *TPM Main Specification, Part 1: Design Principles*, ver. 1.2, Trusted Computing Group, 2003.
8. J. Singh, W. Weber, and A. Gupta, "Splash: Stanford Parallel Applications for Shared-Memory," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 1, 1992, pp. 5–44.

Qian Liu *is a PhD candidate in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has an MSc in computer science from Shanghai Jiao Tong University. Contact him at piano@sjtu.edu.cn.*

Chuliang Weng *is an associate professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a PhD in computer science from Shanghai Jiao Tong University. Contact him at clweng@sjtu.edu.cn.*

Minglu Li *is a professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a PhD in computer science from Shanghai Jiao Tong University. Contact him at li-ml@cs.sjtu.edu.cn.*

Yuan Luo *is a professor in Shanghai Jiao Tong University's Department of Computer Science and Engineering. He has a PhD in probability and mathematical statistics from Nankai University. Contact him at luoyuan@cs.sjtu.edu.cn.*