Eliminating Fine Grained Timers in Xen

Bhanu C. Vattikonda Sambit Das Hovav Shacham

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA
{bvattikonda, skdas, hovav}@cs.ucsd.edu

ABSTRACT

The move to "infrastructure-as-a-service" cloud computing brings with it a new risk: cross-virtual machine side channels through shared physical resources such as the L2 cache. One approach to this risk is to rewrite sensitive code to eliminate the signal. In this paper we consider another approach: weakening malicious virtual machines' ability to receive the signal by eliminating fine-grained timers. Such "fuzzy time" was implemented in 1991 in the VAX security kernel, but it was not clearly applicabile to modern virtual machine managers such as Xen on platforms such as the x86, which exports a cycle counter through the *RDTSC* instruction.

In this paper, we demonstrate that it is possible to modify the *RDTSC* instruction on Xen-virtualized x86 machines, making the timer provided by this instruction substantially more coarse. We perform a thorough evaluation of the impact of modifying this timer on the usability of the system, and we evaluate the limiting point of the timer coarseness.

Our findings open the way to a specific research program for mitigating cloud computing side channels through fuzzy time: (1) What other sources of fine-grained time are available to a malicious VM, and is it possible to degrade them? (2) What distribution of noise should be introduced to *RDTSC* and other timing signals to maximize the effect on malicious VMs while minimizing the effect on legitimate ones? (3) What timing resolution is actually needed to make use of L2 cache side channels?

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized Access

General Terms

Security, Measurement, Experimentation

Keywords

Xen, Side Channels, RDTSC, Cloud Computing

CCSW'11, October 21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

The move to "infrastructure as a service" cloud computing platforms, such as Amazon's EC2 and Microsoft Windows Azure, brings with it the promise of a computing utility: flexible computing that reaps economies of scale. Cloud computing also brings with it new security challenges [4].

Notable among the new security challenges raised by cloud computing is the risk of information disclosure through side channels [12]. To make better use of resources, cloud computing providers multiplex several virtual machines — from different, mutually distrusting clients — on a single physical machine. As Ristenpart et al. showed, an attacker might exploit placement vulnerabilities to achieve coresidence of his VM with a victim VM. With the two VMs sharing physical resources and imperfectly isolated by the virtual machine monitor, the attacker can take advantage of the available side channels to steal private information about computation performed by the victim.

The highest bandwidth — and therefore most dangerous — side channel identified in virtualized environments is the L2 cache. Between processes on the same OS, cache side channels have been shown to allow the exfiltration of cryptographic keys [2, 11, 13]. Virtualization introduces additional noise, but Ristenpart et al. were able to use the L2 cache to measure instantaneous load between coresident VMs.

There are several possible approaches to countering the threat of cache side channels in cloud computing. First, one could reimplement specific routines using algorithms whose memory access patterns do not vary according to secret information like cryptographic key, or introduce noise that would make measurement harder. Tromer, Osvik, and Shamir give a comprehensive discussion of such countermeasures [13]; see also Brickell et al. [3]. Second, one could redesign the cloud computing infrastructure around deterministic execution [1], eliminating timing-based side channels altogether.

1.1 Fuzzy time

In this paper we explore a third approach: limiting the ability of malicious VMs to obtain timing measurements. This approach would leave the side channel in place, but make it hard for the adversary to make use of it. The intuition is that distinguishing between L2 cache hits and misses requires time resolution on the order of tens of nanoseconds (see, e.g., [5]). If high resolution clocks can be eliminated without breaking legacy applications or degrading their performance, and if attackers do not have access to other high quality timing sources, then cache side channels would be

mitigated without rewriting vulnerable programs or constraining attacker behavior.

The idea of making time fuzzier to weaken timing-based channels is not new. Indeed, in 1991 Hu [9] presented just such an approach for the VAX security kernel, which multiplexed several virtual machines running at different classifications. (Hu's goal was to defeat covert channels, not side channels, but an approach that prevents communication of information from a willing transmitter will also prevent the communication from an unwilling one). VAX fuzzy timing randomizes the clock interrupts delivered to VMs. These interrupts, ordinarily delivered every 10 ms, are instead delivered at random intervals sampled from a distribution with mean of 20 ms. In addition, VAX fuzzy time increases the granularity of system time seen by VMs to 100 ms, and synchronizes VM I/O operations with ticks of the fuzzy clock. (For more on fuzzy time, see Gray [7, 8].)

However, it is not clear that this approach translates to current hardware architectures such as the x86 and current virtual machine monitors such as Xen. The x86 exposes a very high resolution clock using the unprivileged *RDTSC* ("read timestamp") instruction. Percival, considering mitigations to AES, is skeptical that fuzzy time is a viable approach on the x86, because applications rely on *RDTSC* for precise time:

> Finally, the traditional method of closing covert timing channels is available: Access to the clock — in this case, the time stamp counter — can be removed. However, this is only an option on single-processor systems: On multi-processor systems, a "virtual" time stamp counter with sufficient precision could be obtained by utilizing a second thread which repeatedly increments a memory location. Even on uniprocessor systems, this approach should not be taken lightly, since many applications expect the time stamp counter to be available, either for profiling purposes, or to be used in combination with a random stream of events (e.g., key presses) as a source of entropy. A somewhat more tolerable approach would limit the frequency with which the time stamp counter could be read — say, to a maximum of four times within any 10000 cycle window — which would be very unlikely to affect any "real" software; but this could only be performed via modifications in the microcode, and it is not clear if the necessary modifications would even be possible given existing architectural limitations. [11]

(We will consider Percival's other objection, that timing sources besides *RDTSC* are available to the attacker, in Section 4.)

1.2 Our findings

In this paper, we show that it is possible, on x86 platforms, to modify the Xen hypervisor to degrade the resolution of *RDTSC*. Based on our modified Xen, we measure the effects of *RDTSC* degradation on real-world workloads using micro- and macro-benchmarks. We find that the system remains stable for *RDTSC* perturbations to a granularity of $2\mu s$ or 4096 cycles.

Beyond this granularity, our paravirtualized testbed system becomes unstable. But, as we discuss in Section 5.2 it is possible to achieve greater degradation in the case of fully virtualized VMs (upto 10M cycles).

Our findings suggest a practical modification to the Xen hypervisor that reduces the quality of timing signal from the x86 cycle counter with a minimal effect on legitimate workloads.

Our findings point the way to a specific research program for mitigating cloud computing side channels:

1. What other sources of fine-grained time are available to a malicious VM, and is it possible to degrade them? (We consider this question briefly in Section 4.)

2. What distribution of noise should be introduced to *RDTSC* and other timing signals to maximize the effect on malicious VMs while minimizing the effect on legitimate ones? (Section 5.1)

3. What timing resolution is actually needed to make use of L2 cache side channels?

Answers to these questions will allow researchers to determine whether fuzzy time is an effective solution to cache side channels in modern cloud computing environments.

We stress that it is our findings in this paper that make possible the research program above. A contrary finding — that perturbations to *RDTSC* on the order of nanoseconds have an effect on legitimate workloads — would have ruled out the entire approach. Whatever other mitigations hypervisors put in place, they would always have to supply fine-grained time to benign and malicious VMs through *RDTSC*.

Though much remains to be done, this paper makes some concrete contributions. We give a demonstration of the possibility of modifying *RDTSC* on Xen-virtualized x86 machines. We perform a thorough evaluation of the impact of modifying this timer on the usability of the system. And we evaluate the limiting point of the coarseness of the timer.

2. EXPERIMENT DESCRIPTION

Our goal was to fuzz the timers available to the guest OS and the applications running in the VM. Once the guest OS does not have access to fine grained timers, we observe the impact fuzzy timers have on applications running in the guest OS.

To modify the granularity of the timers in guest operating system we focus on modifying the value of the *RDTSC* register as seen by the guest operating systems and applications running in the userland of those guest operating systems. This appears to be a reasonable choice because most of the fine grained measurements are done using *RDTSC* instructions. Also, Linux updates the *xtime* variable which maintains the system time using the value of *RDTSC* register. Thus, modifying it is enough to affect the timing related calls in the guest operating systems. We discuss the limitations of this choice in Section 5.

The task of modifying the value of the *RDTSC* register seen by the guest operating system was greatly simplified by the presence of the *softtsc* kernel option in Xen. This kernel option makes the hypervisor trap and emulate the *RDTSC* instruction issued by guest operating system and applications running in the guest VM.

Starting with Xen 4.0 using the *softtsc* option the *RDTSC* instruction can be emulated for both fully virtualized and

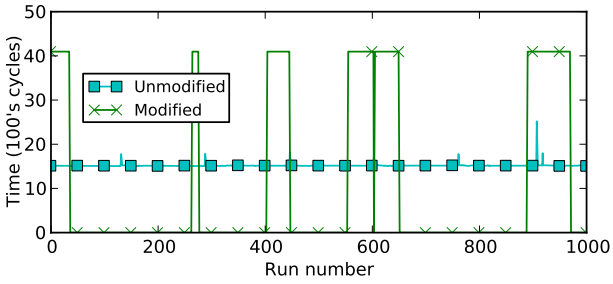

Figure 1: Difference between consecutive *RDTSC* values on unmodified system and a system in which *RDTSC* value follows a step (4096) function

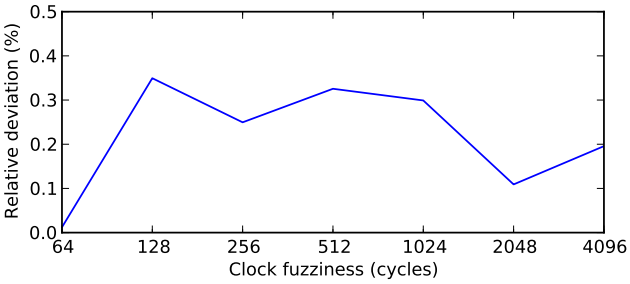

Figure 2: Executing time of FFTW

para-virtualized systems. Earlier versions of Xen did not allow emulation of the *RDTSC* instruction for para-virtualized guest operating systems. Since most cloud service providers like EC2 use para-virtualization, we perform our experiments on para-virtualized guest operating systems. We note that we have performed experiments on fully virtualized guest operating systems with similar results but omit these results due to space constraints.

On using the sofftsc option, Xen traps the *RDTSC* instruction to the textttpv_soft_rdtsc() function. In this function, it stores the value of *rdtscll* into the registers maintained for the guest VM. We modify the *EAX* register, thereby modifying the lower bits of the *RDTSC* counter. We implement the modifications to timer by modifying the value of the *RDTSC* instruction returned by the hypervisor.

The values of *RDTSC* as seen by applications running in the guest operating system running on top of a modified hypervisor show that the changes do in fact influence the timers seen by the guest operating system.

In Figure 1, we show the difference between two consecutive *RDTSC* instructions on the modified and the unmodified Xen system. In the case of unmodified system we see that the time difference between two consecutive *RDTSC* instructions is nearly constant. We then modify the underlying Xen to return *RDTSC* value which is rounded off by 4096 cycles and then plot the difference between consecutive *RDTSC* instructions. As we see in the figure, the difference between consecutive *RDTSC* instructions is mostly 0 and occasionally it is around 4096. This can only be because some times we cross the edge of the step function between to consecutive *RDTSC* instructions.

While we do not present the results here due to lack of space, we note that the changes to the value of the Xen *RDTSC* register also affect the time as seen by typical timing related system calls like *gettimeofday* or *clock_gettime*. Thus, not only *RDTSC*, but other fine grained timers have been affected by modifying the emulated value of *RDTSC* register.

3. PERFORMANCE EVALUATION

We evaluate the performance of typical cloud based applications running in the guest operating system while changing the fuzziness of the timers. Since TCP makes two timing measurements for each packet being sent, it is also ensured that the applications make frequent *RDTSC* measurements. We conducted our experiments on a modest testbed of two machines connected through a 1Gbps Linksys switch. Each machine has a 2.5GHz quad-core processor with 6GB memory running Linux version 2.6.32 and Xen 4.0.1. The guest operating system is para-virtualized in order to achieve the best performance. The guest operating system is configured to use 2GB of RAM and a disk image of 20GB with two dedicated cores.

As discussed in Section 2, we modify the granularity of the clock by modifying the value of the *RDTSC* register read by the guest operating system and the guest applications. The modifications we introduce to the value of this register were all based on step functions. Instead of allowing the guest OS and applications to read the value of *RDTSC* register, the modified Xen would trap and emulate the *RDTSC* instruction and return a value that is rounded off to make the timer coarse. This means that the value of the *RDTSC* register observed by the guest operating system and applications follows a step function. In the following sections we discuss the impact of fuzzy timers on various applications running in the guest operating system.

In all the graphs in this section, the fuzziness introduced in the clock is indicated by the number of cycles the value of the *RDTSC* register is rounded off to. On the machines that we have, a clock fuzz of 4096 cycles is about 2 μs.

3.1 Micro-benchmarks

We begin by evaluating the impact fuzzy timers could have on individual components of the system. In the following sections we show that the performance of CPU, network and disk, all remain unaffected by changes to the granularity of the underlying clock.

3.1.1 *Compute intensive tasks*

The time to complete CPU intensive tasks should not be affected by the granularity of the clock seen by the guest operating system. We see that this is indeed the case by running an out of place forward complex 2D transform of 2048 rows and 4096 columns using a standard C library ([6]). In Figure 2 we show the deviation in time taken for this computation with respect to an unmodified hypervisor.

As expected the granularity of the clock has little impact on the time it takes for a compute intensive job to finish.

3.1.2 *Network throughput*

To evaluate the effect of coarse grain timers on kernel mechanics (TCP engine, scheduling, etc), we ran throughput measurement tests to test any impact the changes to the timers could have on the functioning of the TCP engine. For this experiment, we have a TCP source running on VM_1

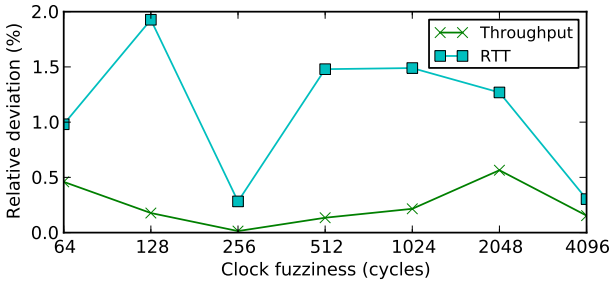

Figure 3: Effect on Throughput and RTT

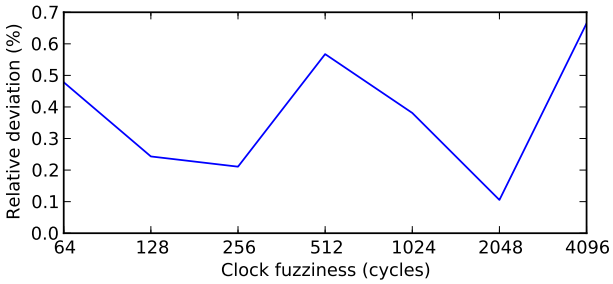

Figure 4: Disk latency observed by the guest VM

sending 1GB data to a sink running on VM_2. We then measure the time it takes for the transfer and calculate the throughput achieved by the flow.

We ran the above experiment twice for each clock granularity. In Figure 3 we show the deviation observed in throughput of the TCP flow relative to the throughput seen in the case of unmodified hypervisor. We can see that the throughput of the flow does not change with increasing clock coarseness (up to a fuzziness of 4096). Thus, applications which are network intensive and make frequent timer calls are unaffected by changes to the clock granularity.

We believe that the throughput of the TCP flows degrades if the fuzziness in the clock is increased. Unfortunately, in the case of para-virtualized VMs we have not been able to fuzz the clock by more than 10000 and hence we do not see the performance degradation in Figure 3. The degradation can however be seen in the case of fully virtualized VMs where the clock can be modified to a greater extent. We discuss this in Section 5.2.

3.1.3 Network latency

Next, we measure the impact of the changes on the latency between the VMs. For this, we use a UDP based ping with VM_1 sending a ping and VM_2 responding immediately to the ping. We then measure the round trip time as seen at VM_1. In Figure 3 we show the deviation in the RTT averaged over 1000 measurements for each clock granularity.

We observe that the latency between the VMs remains unaffected by the changes to the underlying clock. Thus, two important aspects of the network, latency and throughput appear to be unaffected by the changes to the granularity of the clock.

3.1.4 Disk Intensive Workload

We use IOLAT [10] to measure the latency of a disk operation. IOLAT performs a random read from the userspace

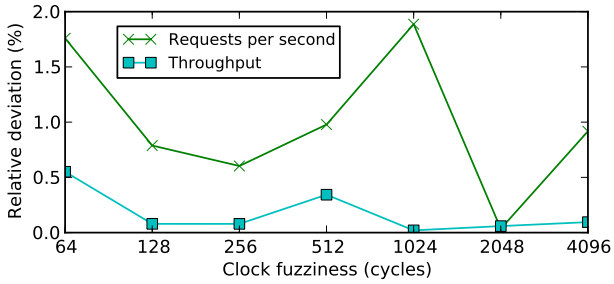

Figure 5: Request/sec handled by apache web server

and measures the time taken for the operation. In Figure 4 we show the deviation in average latency for such a random read operation performed in the guest operating system.

Again, the fuzziness of the timers in the guest operating system does not influence the performance.

3.2 Macro-benchmarks

While the above benchmarks illustrated that the individual components of the system perform normally with clocks operating at coarse granularity, the workloads observed in a cloud based application are very different. In this section we evaluate the performance of the system under workloads which are typically observed in a cloud based application.

3.2.1 Web server performance

Web server is one of the most common applications to be hosted on the cloud platform. We benchmark the performance of an Apache web server running in a VM over the modified hypervisor. We evaluate the number of requests per second that can be handled by the web server for small sized requests and the throughput achieved by the web server when transferring large files.

For these tests, we run the Apache web server on VM_1 and the Apache benchmark suite on VM_2 to observe the performance of the system.

For measuring the requests per second we request a static 10KB html file from the web server using the benchmark suite. Figure 5 shows the deviation in the average requests per second handled by the web server when the benchmark is run with a concurrency of 25, 50 and 75.

The throughput achieved by the server is measured by downloading a 15MB file from the web server using the Apache benchmark suite. In Figure 5 we show the deviation in throughput for different clock granularities. As can be seen in the figure, the throughput achieved by the system remains unaffected by the underlying clock granularity.

3.2.2 All-to-all transfer

Systems like Map-Reduce and Hadoop have a shuffle phase in which state has to be transferred between all the participating nodes. In this phase each node sends data to every other node. We perform a similar shuffle on our testbed between the VMs and measure the time taken for the transfer.

Each VM transfers 1GB of data from it's memory to the other VM. We are interested in the finish times of such a shuffle and show them in Figure 6 and observe that the finish times remain largely unaffected by the granularity of the clock.

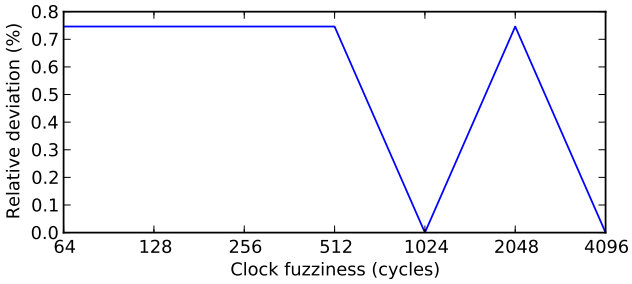

Figure 6: Completion Time for all-to-all transfer

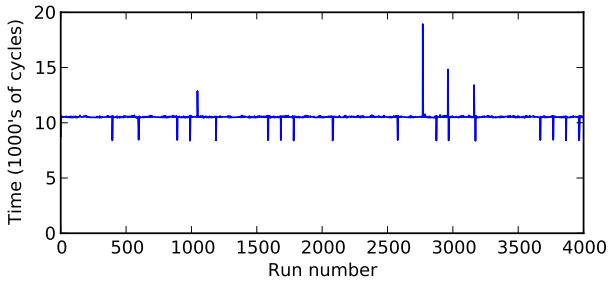

Figure 7: Time taken for executing 1000 NOP's

4. WORKAROUNDS TO *RDTSC*

The focus of our work has been to modify the value of the *RDTSC* register as observed by the guest operating system. While this does eliminate the straightforward ways of making fine grained timing measurements we note that there may be other possible ways for an attacker to make fine grained timing measurements.

First, the attacker could use a pure compute job on a machine under their control, with a similar architecture as the ones in the cloud and note the job's completion time. Once they have such a baseline, the attacker could use such a compute job with core isolation to time the various actions they are interested in. An example of such a scenario we have explored is the use of a constant number of loops with a NOP.

In figure 7 we show the time taken to execute 1000 NOP's in a guest operating system running on top of an unmodified Xen system with core pinning. It can be seen that there is some variation in the time taken to execute 1000 NOP's. Whether this variation is enough to deter an attacker or we need to some how prevent an attacker from making such measurements needs to be explored further.

Without exploring further, we note that the attacker could also use network based measurements to perform fine grained timing measurements.

5. DISCUSSION

5.1 Perturbation function

In this paper the perturbations to the timers that we considered are only step functions. While this was an easy modification to introduce, the attacker can easily observe the perturbation being introduced by waiting for the time when the timer changes. She can then add an offset to the timer by using some computation based measurements as we discuss in Section 4.

To prevent such a possibility we recommend that the perturbation that is introduced have a random behavior. However, the perturbation should still ensure that the value of the *RDTSC* register returned is monotonically increasing. It is easy to see that not doing otherwise would lead to inconsistent system performance. An example of this would be inconsistent file modification times which could affect applications like *make*. One such approach is outlined in Algorithm 1.

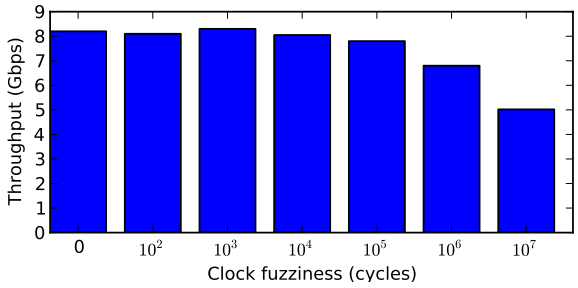

Figure 8: Throughput to localhost in the case of fully virtualized VM runnning on modified Xen

Algorithm 1 Algorithm for suggested change to Xen timer

Algorithm modifyRDTSC

$fuzz = rand()$ // random number between 0, 4096
$t = now - now\%fuzz$ // now is the current time
if $vm.last > t$ **then**
  $regs.eax = vm.last - vm.last\%fuzz$
  $regs.edx = vm.last >> 32$
else
  $regs.eax = now - now\%fuzz$
  $regs.edx = now >> 32$
end if
$vm.last = regs.edx << 32 | regs.eax$

5.2 Para-virtualized vs Fully virtualized VMs

While we show that it is possible to eliminate fine grained timers available to the guest OS by modifying the underlying hypervisor, there are limits to the perturbation that can be introduced. These limits are different for the case of fully virtualized VMs and para-virtualized VMs.

In the case of para-virtualized VMs we have observed that the system becomes unusable (VM instantly hangs) when the *RDTSC* counter is perturbed by more than 10000. This means that the clock can be perturbed upto a few microseconds (10000 cycles). On the other hand, we have observed that a fully virtualized system can handle perturbation of about $10M$. This means that in those cases the clock can operate at millisecond granularity.

While the system continues to be usable, in the case of fully virtualized VMs we observe an interesting pattern in the performance of network throughput. In Figure 8 we see that upto the granularity of 10's of μs the performance remains unaffected by the clock fuzz, it then begins to drop before the system becomes unusable.

We believe that the main reason for this difference in behavior is the difference in the clock sources used by the fully virtualized VM and the para-virtualized VM. The latest *pvops* kernel used as the guest OS in para-virtualized mode uses a *xen* clock source while the unmodified Linux running in fully virtualized mode uses *tsc* clock source. The *pvops* kernel falls back to the *xen* clock source even if we set kernel clock source option to be *tsc*. Whether the *pvops* kernel can be forced to use the *tsc* clocksource remains to be explored further.

6. CONCLUSION

In this paper we show that it is possible to degrade the granularity of the clock available to guest operating system without affecting the performance of typical applications. Further research needs to be done to explore other possible timers available to attackers, ways to degrade them and the nature of the noise that these degradations should introduce.

Acknowledgments

We thank Dan Boneh and Vitaly Shmatikov for helpful discussions and the CCSW reviewers for their comments.

This material is based upon work supported by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

7. REFERENCES

[1] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In A. Perrig and R. Sion, editors, *Proceedings of CCSW 2010*. ACM Press, Oct. 2010.

[2] D. J. Bernstein. Cache-timing attacks on AES, Apr. 2005. Online: `http://cr.yp.to/papers.html#cachetiming`.

[3] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. `http://eprint.iacr.org/`.

[4] Y. Chen, V. Paxson, and R. Katz. What's new about cloud computing security? Technical Report UCB/EECS-2010-5, UC Berkeley Department of EECS, jan 2010. Online: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.pdf`.

[5] J. Dean. Designs, lessons and advice from building large distributed systems. Invited talk at LADIS 2009, Oct. 2009. Online: `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

[6] FFTW. `http://www.fftw.org/`.

[7] J. W. Gray. On analyzing the bus-contention channel under fuzzy time. In C. Meadows, editor, *Proceedings of CSFW 1993*, pages 3–9. IEEE Computer Society, June 1993.

[8] J. W. Gray. Countermeasures and tradeoffs for a class of covert timing channels. Technical Report HKUST-CS94-18, Hong Kong Uuiversity of Science and Technology, 1994. Online: `http://hdl.handle.net/1783.1/25`.

[9] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of IEEE Security and Privacy ("Oakland") 1991*, pages 8–20. IEEE Computer Society, May 1991.

[10] IOLAT disk latency measure. `http://pedro.larroy.com/devel/iolat/`.

[11] C. Percival. Cache missing for fun and profit. Presented at BSDCan 2005, May 2005. Online: `http://www.daemonology.net/papers/htt.pdf`.

[12] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In S. Jha and A. Keromytis, editors, *Proceedings of CCS 2009*, pages 199–212. ACM Press, Nov. 2009.

[13] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, Jan. 2009.