Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

11-2014

StopWatch: A Cloud Architecture for Timing Channel Mitigation

Peng Li

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Michael K Reiter

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Information Security Commons, and the Systems Architecture Commons

Citation

Li, Peng; GAO, Debin; and Reiter, Michael K. StopWatch: A Cloud Architecture for Timing Channel Mitigation. (2014). *ACM Transactions on Information and System Security (TISSEC)*. 17, (2),.
Available at: https://ink.library.smu.edu.sg/sis_research/2525

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

StopWatch: A Cloud Architecture for Timing Channel Mitigation

PENG LI, University of North Carolina at Chapel Hill
DEBIN GAO, Singapore Management University
MICHAEL K. REITER, University of North Carolina at Chapel Hill

This article presents StopWatch, a system that defends against timing-based side-channel attacks that arise from coresidency of victims and attackers in infrastructure-as-a-service clouds. StopWatch triplicates each cloud-resident guest virtual machine (VM) and places replicas so that the three replicas of a guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. StopWatch uses the timing of I/O events at a VM's replicas collectively to determine the timings observed by each one or by an external observer, so that observable timing behaviors are similarly likely in the absence of any other individual, coresident VMs. We detail the design and implementation of StopWatch in Xen, evaluate the factors that influence its performance, demonstrate its advantages relative to alternative defenses against timing side channels with commodity hardware, and address the problem of placing VM replicas in a cloud under the constraints of StopWatch so as to still enable adequate cloud utilization.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms: Security, Design

Additional Key Words and Phrases: Timing channels, side channels, clouds, virtualization, replication

ACM Reference Format:
Peng Li, Debin Gao, and Michael K. Reiter. StopWatch: A cloud architecture for timing channel mitigation *ACM Trans. Inf. Syst. Secur.* 17, 2, Article 8 (November 2014), 28 pages.
DOI:http://dx.doi.org/10.1145/2670940

8

1. INTRODUCTION

Implicit timing-based information flows threaten the use of clouds for very sensitive computations. In an infrastructure-as-a-service (IaaS) cloud, such an attack could be mounted by an attacker submitting a virtual machine (VM) to the cloud that times the duration between events that it can observe, to make inferences about a *victim* VM with which it is running simultaneously on the same host but otherwise cannot access. Such "access-driven" attacks [Zhang et al. 2012b] were first studied in the context of timing-based *covert channels*, in which the victim VM is infected with a Trojan horse that intentionally signals information to the attacker VM by manipulating the timings that the attacker VM observes. Of more significance in modern cloud environments, however, are timing-based *side channels*, which leverage the same principles to attack an uninfected but oblivious victim VM (e.g., Ristenpart et al. [2009]; Zhang et al. [2012b]).

In this article, we propose an approach for defending against these timing attacks and a system, called StopWatch, that implements this method for IaaS clouds. A timing

This work was supported in part by NSF grants 0910483 and 1330599, the Science of Security Lablet at North Carolina State University, and grants from IBM and VMWare.
Authors' addresses: P. Li, VMWare, Palo Alto, CA; D. Gao, School of Information Systems, Singapore Management University, Singapore; M. K. Reiter (corresponding author), Department of Computer Science, University of North Carolina, Chapel Hill, NC; email: reiter@cs.unc.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
2014 Copyright is held by the author/owner(s).
1094-9224/2014/11-ART8 $15.00
DOI:http://dx.doi.org/10.1145/2670940

side channel can arise whenever an attacker VM uses an event sequence it observes to "time" another independent event sequence that might reflect the victim VM's behavior [Wray 1991]. StopWatch is thus designed to systematically remove independence of observable event sequences where possible, first by making all real-time clocks accessible from a guest VM to be determined instead by the VM's own execution.

To address event sequences on which it cannot intervene this way, namely, for input/output (I/O) events, StopWatch alters I/O timings observed by the attacker VM to mimic those of a *replica* attacker VM that is not coresident with the victim. Since StopWatch cannot identify attackers and victims a priori, realizing this intuition in practice requires replicating each VM on multiple hosts and enforcing that the replicas are coresident with nonoverlapping sets of (replicas of) other VMs so that, in particular, at most one attacker VM replica is coresident with a replica of the victim VM. StopWatch then delivers any I/O event to each VM replica at a time determined by "microaggregating" the delivery times planned by the VMMs hosting those replicas. Specifically, StopWatch uses three replicas per VM that coreside with nonoverlapping sets of (replicas of) other VMs and microaggregates the timing of I/O events by taking their median across all three replicas. (Two replicas per VM seems not to be enough: one might be coresident with its victim, and by symmetry, its I/O timings would necessarily influence the timings imposed on the pair.) Even if the median timing of an I/O event is that which occurred at an attacker replica that is coresident with a victim replica, timings both below and above the median occurred at attacker replicas that do not coreside with the victim.

We detail the implementation of StopWatch in Xen, specifically to intervene on all real-time clocks and, notably, to enforce this median behavior on "clocks" available via the I/O subsystem (e.g., network interrupts). Moreover, for a uniprocessor VM (i.e., one limited to using only a single virtual CPU, even when running on a physical platform with multiple physical CPUs), StopWatch enforces deterministic execution across all of the VM's replicas, making it impossible for an attacker VM to utilize other internally observable clocks and ensuring the same outputs from the VM replicas. By applying the median principle to the timing of these outputs, StopWatch further interferes with inferences that an observer external to the cloud could make on the basis of output timings.

We evaluate the performance of our StopWatch prototype for supporting Web service (file downloads) and various types of computations. Our analysis shows that the latency overhead of StopWatch is less than 2.8×, even for network-intensive applications. We also identify adaptations to a service that can vastly increase its performance when run over StopWatch, for example, making file download over StopWatch competitive with file download over unmodified Xen. For computational benchmarks, the latency induced by StopWatch is less than 2.3× and is directly correlated with their amounts of disk I/O. Overall, the latency overhead of StopWatch is qualitatively similar to other modern systems that use VM replication for other reasons (e.g., Cully et al. [2008]). Moreover, we compare StopWatch to other defenses against timing side-channel attacks, namely, adding uniformly random noise to event timings or running VMs on shared hardware in a time-sliced fashion, and demonstrate StopWatch's benefits over them.

We also study the impact of StopWatch on cloud utilization, that is, how many guest VMs can be simultaneously executed on an infrastructure of n machines, each with a capacity of c guest VMs, under the constraint that the three replicas for each guest VM coreside with nonoverlapping sets of (replicas of) other VMs. We show that for any $c \leq \frac{n-1}{2}$, $\Theta(cn)$ guest VMs (three replicas of each) can be simultaneously executed; we also identify practical algorithms for placing replicas to achieve this bound. We

extend this result to $\Theta(\frac{cn}{d_{max}})$ guest VMs when guest VMs can place different demands, up to d_{max}, on machine resources of capacity c. These results distinguish StopWatch from the alternative of simply running each guest VM on a separate computer, which permits simultaneous execution of only n guest VMs.

To summarize, our contributions are as follows: First, we introduce a novel approach for defending against access-driven timing side-channel attacks in infrastructure-as-a-service (IaaS) compute clouds that leverages replication of guest VMs with the constraint that the replicas of each guest VM coreside with nonoverlapping sets of (replicas of) other VMs. The median timings of I/O events across the three guest VM replicas are then imposed on these replicas to interfere with their use of event timings to extract information from a victim VM with which one is coresident. Second, we detail the implementation of this strategy in Xen, yielding a system called StopWatch, and evaluate the performance of StopWatch on a variety of workloads. This evaluation sheds light on the features of workloads that most impact the performance of applications running on StopWatch and how they can be adapted for best performance. We further extend this evaluation with a comparison to other plausible alternatives for defending holistically against access-driven timing side-channel attacks, such as adding random noise to the observable timing of events or running VMs on shared hardware in a time-sliced fashion. Third, we show how to place replicas under the constraints of StopWatch to utilize a cloud infrastructure more effectively than running each guest VM in isolation.

The rest of this article is structured as follows. We describe related work in Section 2. We provide an overview of the design of StopWatch in Section 3 and detail how we address classes of internal "clocks" used in timing attacks in Section 4 and Section 5. In Section 6, we then discuss how StopWatch extends to address richer attacks involving collaborators external to the cloud or collaborative attacker VMs. We evaluate performance of our StopWatch prototype in Section 7. We extend this evaluation to provide a comparison to other holistic timing side-channel defenses in Section 8. Section 9 treats the replica placement problem that would be faced by cloud operators using StopWatch, and we conclude in Section 10.

2. RELATED WORK

2.1. Timing Channel Defenses

Defenses against information leakage via timing channels are diverse, taking numerous different angles on the problem. Research on type systems and security-typed languages to eliminate timing attacks offers powerful solutions (e.g., Agat [2000], Zdancewic and Myers [2003], Zhang et al. [2012a]), but this work is not immediately applicable to our goal here, namely, adapting an existing virtual machine monitor (VMM) to support practical mitigation of timing channels today. Other research has focused on the elimination of timing side channels within cryptographic computations (e.g., [Tromer et al. 2010]) or as enabled by specific hardware components (e.g., [Kim et al. 2012; Raj et al. 2009]), but we seek an approach that is comprehensive.

Askarov et al. [2010] distinguish between *internal* timing channels that involve the implicit or explicit measurement of time from within the system, and *external* timing channels that involve measuring the system from the point of view of an external observer. Defenses for both internal (e.g., Agat [2000], Hu [1991], Vattikonda et al. [2011], Zdancewic and Myers [2003]) and external (e.g., Askarov et al. [2010], Giles and Hajek [2002], Haeberlen et al. [2011], Kang and Moskowitz [1993], Zhang et al. [2011]) timing channels have received significant attention individually, though to our knowledge, StopWatch is novel in addressing access-driven timing channels through a combination of both techniques. StopWatch incorporates internal defenses to

interfere with an attacker's use of real-time clocks or "clocks" that it might derive from the I/O subsystem. In doing so, StopWatch imposes determinism on uniprocessor VMs and then uses this feature to additionally build an effective external defense against such attacker VMs.

StopWatch's internal and external defense strategies also differ individually from prior work, in interfering with timing channels by allowing replicas (in the internal defenses) and external observers (in the external defenses) to observe only median I/O timings across the three replicas. The median offers several benefits over the alternative of obfuscating event timings by adding random noise (without replicating VMs): to implement random noise, a distribution from which to draw the noise must be chosen without reference to an execution in the absence of the victim—that is, how the execution "should have" looked—so ensuring that the chosen noise distribution is sufficient to suppress all timing channels can be quite difficult. StopWatch uses replication and careful replica placement (in terms of the other VMs with which each replica coresides) exactly to provide such a reference. Moreover, we show that the median permits the delays incurred by the system to scale better than uniformly random noise allows for the same protection, as the distinctiveness of victim behavior increases.

2.2. Replication

To our knowledge, StopWatch is novel in utilizing replication for timing channel defense. That said, replication has a long history that includes techniques similar to those we use here. For example, state-machine replication to mask Byzantine faults [Schneider 1990] ensures that correct replicas return the same response to each request so that this response can be identified by "vote" (a technique related to one employed in StopWatch; see Sections 3 and 6.1). To ensure that correct replicas return the same responses, these systems enforce the delivery of requests to replicas in the same order; moreover, they typically assume that replicas are deterministic and process requests in the order they are received. Enforcing replica determinism has also been a focus of research in (both Byzantine and benignly) fault-tolerant systems; most (e.g., Basile et al. [2006], Borg et al. [1989], Narasimhan et al. [1999]), but not all (e.g., [Bressoud and Schneider 1996]), do so at other layers of the software stack than StopWatch does.

More fundamentally, to our knowledge all prior systems that enforce timing determinism across replicas permit one replica to dictate timing-related events for the others, which does not suffice for our goals: that replica could be the one coresident with the victim, and so permitting it to dictate timing-related events would simply "copy" the information it gleans from the victim to the other replicas, enabling that information to then be leaked out of the cloud. Rather, by forcing the timing of events to conform to the median timing across three VM replicas, at most one of which is coresident with the victim, the enforced timing of each event is either the timing of a replica not coresident with the victim or else between the timing of two replicas that are not coresident with the victim. This strategy is akin to ones used for Byzantine fault-tolerant clock synchronization (e.g., see Schneider [1987, Section 5.2]) or sensor replication (e.g., see Schneider [1990, Section 5.1]), though we use it here for information hiding (versus integrity).

Aside from replication for fault tolerance, replication has been explored to detect server penetration [Cox et al. 2006; Gao et al. 2005, 2009; Nguyen-Tuong et al. 2008]. These approaches purposely employ diverse replica codebases or data representations so as to reduce the likelihood of a single exploit succeeding on multiple replicas. Divergence of replica behavior in these approaches is then indicative of an exploit succeeding on one but not others. In contrast to these approaches, StopWatch leverages

(necessarily) *identical* guest VM replicas to address a different class of attacks (timing side channels) than replica compromise.

Research on VM execution *replay* (e.g., Xu et al. [2007], Dunlap et al. [2008]) focuses on recording nondeterministic events that alter VM execution and then coercing these events to occur the same way when the VM is replayed. The replayed VM is a replica of the original, albeit a temporally delayed one, and so this can also be viewed as a form of replication. StopWatch similarly coerces VM replicas to observe the same event timings, but again, unlike these timings being determined by one replica (the original), they are determined collectively using median calculations, so as to interfere with one attacker VM replica that is coresident with the victim from simply propagating its timings to all replicas. That said, the state of the art in VM replay (e.g., Dunlap et al. [2008]) addresses multiprocessor VM execution, which our present implementation of StopWatch does not. StopWatch could be extended to support multiprocessor execution with techniques for deterministic multiprocessor scheduling (e.g., Devietti et al. [2010]). Mechanisms for enforcing deterministic execution through O/S-level modifications (e.g., Aviram et al. [2010]) are less relevant to our goals, as they are not easily used by an IaaS cloud provider that accepts arbitrary VMs to execute.

3. DESIGN

Our design is focused on infrastructure-as-a-service (IaaS) clouds that accept virtual machine images, or "guest VMs," from customers to execute. Amazon EC2 (`http://aws.amazon.com/ec2/`) and Rackspace (`http://www.rackspace.com/`) are example providers of public IaaS clouds. Given the concerns associated with side-channel attacks in cloud environments (e.g., Ristenpart et al. [2009]; Zhang et al. [2012b]), we seek to develop virtualization software that would enable a provider to construct a cloud that offers substantially stronger assurances against leakage via timing channels. This cloud might be a higher assurance offering that a provider runs alongside its normal cloud (while presumably charging more for the greater assurance it offers) or a private cloud with substantial assurance needs (e.g., run by and for an intelligence or military community).

3.1. Threat Model

Our threat model is a customer who submits *attacker VMs* for execution that are designed to employ timing side channels. We presume that the attacker VM is designed to extract information from a particular victim VM, versus trying to learn general statistics about the cloud, such as its average utilization. We assume that access controls prevent the attacker VMs from accessing victim VMs directly or from escalating their own privileges in a way that would permit them to access victim VMs. The cloud's virtualization software (in our case, Xen and our extensions thereof) is trusted.

According to Wray [1991], to exploit a timing channel, the attacker VM measures the timing of observable events using a *clock* that is independent of the timings being measured. While the most common such clock is real time, a clock can be any sequence of observable events. With this general definition of "clock," a timing attack simply involves measuring one clock using another. Wray identified four possible clock sources in conventional computers [1991].

— TL. The "CPU instruction-cycle clock" (e.g., a clock constructed by executing a simple timing loop).
— Mem. The memory subsystem (e.g., data/instruction fetches).
— IO. The I/O subsystem (e.g., network, disk, and DMA interrupts).
— RT. Real-time clocks provided by the hardware platform (e.g., time-of-day registers).

3.2. Defense Strategy

StopWatch is designed to interfere with the use of IO and RT clocks and, for uniprocessor VMs, TL, or Mem clocks, for timing attacks. (As discussed in Section 2, extension to multiprocessor VMs is a topic of future work.) IO and RT (especially RT) clocks are an ingredient in every timing side-channel attack in the research literature that we have found, undoubtedly because real time is the most intuitive, independent, and reliable reference clock for measuring another clock. So, intervening on these clocks is of paramount importance. Moreover, the way StopWatch does so forces the scheduler in a uniprocessor guest VM to behave deterministically, interfering with attempts to use TL or Mem clocks.

More specifically, to interfere with IO clocks, StopWatch replicates each attacker VM (i.e., every VM, since we do not presume to know which ones are attacker VMs) threefold so that the three replicas of a guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. Then, when determining the timing with which an IO event is made available to each replica, the median timing value of the three is adopted. StopWatch addresses RT clocks by replacing a VM's view of real time with a *virtual* time that depends on the VM's own progress [Popek and Kline 1974].

A side-effect of how StopWatch addresses IO and RT clocks is that it enforces deterministic execution of uniprocessor attacker VM replicas, also disabling its ability to use TL or Mem clocks. These mechanisms thus deal effectively with internal observations of time, but it remains possible that an external observer could glean information from the real-time duration between its receipt of packets sent by the attacker VM. To interfere with this timing channel, we emit packets to an external observer with timing dictated by, again, the median timing of the three VM replicas.

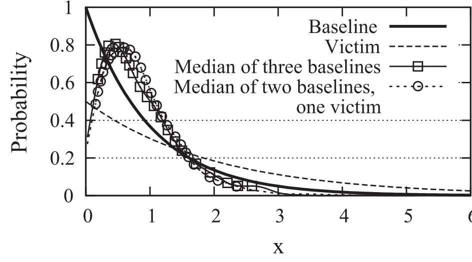
3.3. Justification for the Median

Permitting only the median timing of an IO event to be observed limits the information that an attacker VM can glean from being colocated with a victim VM of interest, because the distribution of the median timings substantially dampens the visibility of a victim's activities.

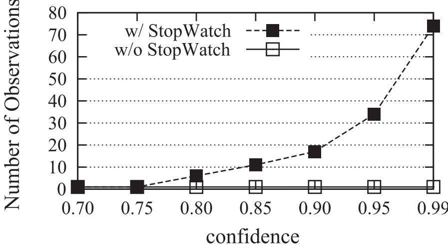
To see why, consider a victim VM that induces observable timings that are exponentially distributed with rate λ', versus a baseline (i.e., non-victim) exponential distribution with rate $\lambda > \lambda'$.[1] (We will broaden our discussion to other distributions later.) Figure 1(a) plots example distributions of the attacker VMs' observations under StopWatch when an attacker VM is coresident with the victim ("Median of two baselines, one victim") and when attacker VM is not ("Median of three baselines"). This figure shows that these median distributions are quite similar, even when λ is substantially larger than λ'; for example, $\lambda = 1$ and $\lambda' = 1/2$ in the example in Figure 1(a). In this case, to even reject the null hypothesis that the attacker VM is not coresident with the victim using a χ-square test, the attacker can do so with high confidence in the absence of StopWatch with only a single observation, but doing so under StopWatch requires almost two orders of magnitude more (Figure 1(b)). This improvement becomes even more pronounced if λ and λ' are closer; the case $\lambda = 1$, $\lambda' = 10/11$ is shown in Figure 1(c).

In terms of the number of observations needed to extract meaningful information from the victim VM, this assessment is very conservative, since the attacker would face numerous pragmatic difficulties that we have not modeled here [Zhang et al. 2012b].

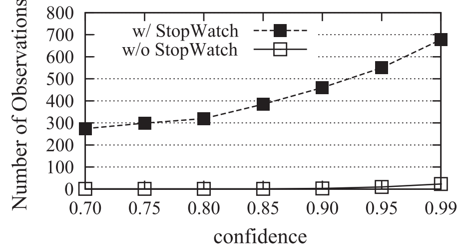
[1] It is not uncommon to model packet interarrival time, for example, using an exponential distribution (e.g., Karagiannis et al. [2004]).



(a) Distribution of median; $\lambda' = 1/2$.



(b) Observations needed to detect victim; $\lambda' = 1/2$.



(c) Observations needed to detect victim; $\lambda' = 10/11$.

Fig. 1. Justification for median; baseline distribution Exp(λ), $\lambda = 1$, and victim distribution Exp(λ').

But even this simple example shows the power of disclosing only median timings of three VM replicas, and in Section 5.2 we will repeat this illustration using actual message traces.

This illustration of the benefits of allowing only the median timing of an IO event to be observed by an attacker is not specific to timing behaviors that are exponentially distributed. Instead, it generalizes to any distribution. To make this clear, let $X_{r:m}$ denote the random variable that takes on the value of the rth smallest of the m values obtained by sampling random variables $X_1 \ldots X_m$. Let $F_i(x)$ denote the CDF of X_i (i.e., $F_i(x) = \mathbb{P}(X_i \leq x)$), and let $F_{r:m}(x)$ denote the CDF of $X_{r:m}$. The security of StopWatch hinges on the distribution of the median $X_{2:3}$ of three independent random variables X_1, X_2, X_3 defined as the difference in virtual times (or, in the case of an external observer, real times) between two subsequent IO events.

Specifically, due to the construction of StopWatch, the adversary is relegated to learning information from the difference between (i) the CDF $F_{2:3}(x)$ for random variables X_1, X_2, X_3 corresponding to attacker VM replicas that are not coresident with a victim VM of interest, and (ii) the CDF $F'_{2:3}(x)$ for random variables X'_1, X_2, X_3, where X'_1 corresponds to an attacker VM that is coresident with the victim VM of interest. An example measure of the distance between two CDFs $F(x)$ and $\hat{F}(x)$ is their Kolmogorov-Smirnov distance [Deza and Deza 2006, p. 179], defined as $D(F, \hat{F}) = \max_x \left| F(x) - \hat{F}(x) \right|$.

The following theorem shows that adopting the median microaggregation function can only interfere with the adversary's goal.

THEOREM 1. *If the distributions of X_2 and X_3 are overlapping (i.e., for no x is $F_2(x) = 0$ and $F_3(x) = 1$, or $F_2(x) = 1$ and $F_3(x) = 0$), then $D(F_{2:3}, F'_{2:3}) < D(F_1, F'_1)$.*

PROOF. Due to well-known results in order statistics (e.g., Güngör et al. [2009, Result 2.4]),[2]

$$F_{r:m}(x) = \sum_{\ell=r}^{m}(-1)^{\ell-r}\binom{\ell-1}{r-1}\sum_{\substack{I\subseteq\{1...m\}:\\ |I|=\ell}}\prod_{i\in I}F_i(x).$$

In particular,

$$F_{2:3}(x) = F_1(x)F_2(x) + F_1(x)F_3(x) + F_2(x)F_3(x) - 2F_1(x)F_2(x)F_3(x),$$
$$F'_{2:3}(x) = F'_1(x)F_2(x) + F'_1(x)F_3(x) + F_2(x)F_3(x) - 2F'_1(x)F_2(x)F_3(x),$$

where $F'_1(x)$ represents the CDF of X'_1. So,

$$D(F_{2:3}, F'_{2:3}) = \max_x \left|[F_2(x) + F_3(x) - 2F_2(x)F_3(x)]\,[F_1(x) - F'_1(x)]\right|.$$

Noting that $D(F_1, F'_1) = \max_x |F_1(x) - F'_1(x)|$, it suffices to show that $|F_2(x) + F_3(x) - 2F_2(x)F_3(x)| < 1$ for all x. However, since $F_2(x) \in [0, 1]$ and $F_3(x) \in [0, 1]$ for all x, $|F_2(x) + F_3(x) - 2F_2(x)F_3(x)| \leq 1$ and, moreover, equals 1 only if for some x, one of $F_2(x)$ and $F_3(x)$ is 1 and the other is 0. This last case is precluded by the theorem. □

In the limit, when the distributions of X_2 and X_3 overlap exactly, we get a much stronger result.

THEOREM 2. *If X_2 and X_3 are identically distributed, then* $D(F_{2:3}, F'_{2:3}) \leq \frac{1}{2}D(F_1, F'_1)$.

PROOF. In this case, $F_2 = F_3$, and so

$$|F_2(x) + F_3(x) - 2F_2(x)F_3(x)|$$

reaches its maximum value of $\frac{1}{2}$ at the value x yielding $F_2(x) = F_3(x) = \frac{1}{2}$. □

4. RT CLOCKS

Real-time clocks provide reliable and intuitive reference clocks for measuring the timings of other events. In this section, we describe the high-level strategy taken in StopWatch to interfere with their use for timing channels and detail the implementation of this strategy in Xen with hardware-assisted virtualization (HVM).

4.1. Strategy

The strategy adopted in StopWatch to interfere with a VM's use of real-time clocks is to virtualize these real-time clocks so that their values observed by a VM are a deterministic function of the VM's instructions executed so far [Popek and Kline 1974]. That is, after the VM executes *instr* instructions, the virtual time observed from within the VM is

$$virt(instr) \leftarrow slope \times instr + start. \tag{1}$$

To determine *start* at the beginning of VM replica execution, the VMMs hosting the VM's replicas exchange their current real times; *start* is initially set to the median of these values. *slope* is initially set to a constant determined by the tick rate of the machines on which the replicas reside.

[2] This equation assumes each $F_i(x)$ is continuous. See Güngör et al. [2009] for the case when some $F_i(x)$ is not continuous.

Optionally, the VMMs can adjust *start* and *slope* periodically, for example, after the replicas execute an "epoch" of I instructions, to coarsely synchronize *virt* and real time. For example, after the kth epoch, each VMM can send to the others the duration D_k over which its replica executed those I instructions and its real time R_k at the end of that duration. Then, the VMMs can select the median real time R_k^* and the duration D_k^* from that same machine and reset

$$start_{k+1} \leftarrow virt_k(I)$$
$$slope_{k+1} \leftarrow \arg\min_{v \in [\ell, u]} \left| \frac{R_k^* - virt_k(I) + D_k^*}{I} - v \right|$$

for a preconfigured constant range $[\ell, u]$, to yield the formula for $virt_{k+1}$.[3] The use of ℓ and u ensures that $slope_{k+1}$ is not too extreme, and, if $\ell > 0$, that $slope_{k+1}$ is positive. In this way, $virt_{k+1}$ should approach real time on the computer contributing the median real time R_k^* over the next I instructions, assuming that the machine and VM workloads stay roughly the same. Of course, the smaller I-values are, the more *virt* follows real time and so poses the risk of becoming useful in timing attacks. So, *virt* should be adjusted only for tasks for which coarse synchronization with real time is important and then only with large I values.

4.2. Implementation in Xen

Real-ime clocks on a typical x86 platform include timer interrupts and various hardware counters. Closely related to these real-time clocks is the timestamp counter register, which is accessed using the `rdtsc` instruction and stores a count of processor ticks since reset.

4.2.1. Timer Interrupts. Operating systems typically measure the passage of time by counting timer interrupts; that is, the operating system sets up a hardware device to interrupt periodically at a known rate, such as 100 times per second [VMWare 2010]. There are various such hardware devices that can be used for this purpose. Our current implementation of StopWatch assumes the guest VM uses a Programmable Interval Timer (PIT) as its timer interrupt source, but our implementation for other sources would be similar. The StopWatch VMM generates timer interrupts for a guest on a schedule dictated by that guest's *virtual* time *virt*, as computed in Eq. (1). To do so, it is necessary for the VMM to be able to track the instruction count *instr* executed by the guest VM.

In our present implementation, StopWatch uses the guest *branch count* for *instr*, that is, keeping track only of the number of branches that the guest VM executes. Several architectures support hardware branch counters, but these are not sensitive to the multiplexing of multiple guests onto a single hardware processor and so continue to count branches regardless of the guest that is currently executing. So, to track the branch count for a guest, StopWatch implements a *virtualized* branch counter for each guest.

A question is when to inject each timer interrupt. Intel VT augments IA-32 with two new forms of CPU operations: virtual machine extensions (VMX) root operation and VMX non-root operation [Uhlig et al. 2005]. While the VMM uses root operation, guest VMs use VMX non-root operation. In non-root operation, certain instructions and events cause a *VM exit* to the VMM so that the VMM can emulate those instructions

[3] In other words, if $(R_k^* - virt_k(I) + D_k^*)/I \in [\ell, u]$, then this value becomes $slope_{k+1}$. Otherwise, either ℓ or u does, whichever is closer to $(R_k^* - virt_k(I) + D_k^*)/I$.

or deal with those events. Once completed, control is transferred back to the guest VM via a *VM entry*. The guest then continues running as if it had never been interrupted.

VM exits give the VMM the opportunity to inject timer interrupts into the guest VM as the guest's virtual time advances. However, so that guest VM replicas observe the same timer interrupts at the same points in their executions, StopWatch injects timer interrupts only after VM exits that are caused by guest execution. Other VM exits can be induced by events external to the VM, such as hardware interrupts on the physical machine; these would generally occur at different points during the execution of the guest VM replicas but will not be visible to the guest [Intel 2011, Section 29.3.2]. For VM exits caused by guest VM execution, the VMM injects any needed timer interrupts on the next VM entry.

4.2.2. `rdtsc` Calls and CMOS RTC Values. Another way for a guest VM to measure time is via `rdtsc` calls. Xen already emulates the return values to these calls. More specifically, to produce the return value for a `rdtsc` call, the Xen hypervisor computes the time passed since guest reset using its real-time clock, and then this time value is scaled by a constant factor. StopWatch replaces this use of a real-time clock with the guest's virtual clock (Eq. (1)).

A virtualized real-time clock (RTC) is also provided to HVM guests in Xen; this provides time to the nearest second for the guest to read. The virtual RTC gets updated by Xen using its real-time clock. StopWatch responds to requests to read the RTC using the guest's virtual time.

4.2.3. Reading Counters. The guest can also observe real time from various hardware counters, for example, the PIT counter, which repeatedly counts down to zero (at a pace dictated by real time) starting from a constant. These counters, too, are already virtualized in modern VMMs, such as Xen. In Xen, these return values are calculated using a real-time clock; StopWatch uses the guest virtual time, instead.

5. IO CLOCKS

IO clocks are typically network, disk, and DMA interrupts. (Other device interrupts, such as keyboards, mice, graphics cards, etc., are typically not relevant for guest VMs in clouds.) We outline our strategy for mitigating their use to implement timing channels in Section 5.1, and then in Section 5.2, we describe our implementation of this strategy in StopWatch.

5.1. Strategy

The method described in Section 4 for dealing with RT clocks by introducing virtual time provides a basis for addressing sources of IO clocks. A component of our strategy for doing so is to synchronize I/O events across the three replicas of each guest VM in virtual time so that every I/O interrupt occurs at the same virtual time at all replicas. Among other things, this synchronization will force uniprocessor VMs to execute deterministically, but it alone will not be enough to interfere with IO clocks; it is also necessary to prevent the timing behavior of one replica's machine from imposing I/O interrupt synchronization points for the others, as discussed in Section 2–3. This is simpler to accomplish for disk accesses and DMA transfers, since replica VMs initiate these themselves, and so we will discuss this case first. The more difficult case of network interrupts, where we explicitly employ median calculations to dampen the influence of any one machine's timing behavior on the others, will then be addressed.

5.1.1. Disk and DMA Interrupts. The replication of each guest VM at start time includes replicating its entire disk image, and so any disk blocks available to one VM replica will be available to all. By virtue of the fact that (uniprocessor) VMs execute

deterministically in StopWatch, replicas will issue disk and DMA requests at the same virtual time. Upon receiving such a request from a replica at time V, the VMM adds an offset Δ_d to determine a "delivery time" for the interrupt, that is, at virtual time $V + \Delta_d$, and initiates the corresponding I/O activities (disk access or DMA transfer). The offset Δ_d must be large enough to ensure that the data transfer completes by the virtual delivery time. Once the virtual delivery time has been determined, the VMM simply waits for the first VM exit caused by the guest VM (as in Section 4.2) that occurs at a virtual time at least as large as this delivery time. The VMM then injects the interrupt prior to the next VM entry of the guest. This interrupt injection also includes copying the data into the address space of the guest so as to prevent the guest VM from polling for the data in advance of the interrupt to create a form of clock (e.g., see [Hu 1991, Section 4.2.2]).

5.1.2. Network Interrupts. Unlike the initiation of disk accesses and DMA transfers, the activity giving rise to a network interrupt, namely, the arrival of a network packet that is destined for the guest VM, is not synchronized in virtual time across the three replicas of the guest VM. So, the VMMs on the three machines hosting these replicas must coordinate to synchronize the delivery of each network interrupt to the guest VM replicas. To prevent the timing of one from dictating the delivery time at all three, these VMMs exchange proposed delivery times and select the median, as discussed in Section 3. To solicit proposed timings from the three, it is necessary, of course, that the VMMs hosting the three replicas all observe each network packet. So, StopWatch replicates every network packet to all three computers hosting replicas of the VM for which the packet is intended. This is done by a logically separate "ingress node" that we envision residing on a dedicated computer in the cloud. (Of course, there need not be only one such ingress for the whole cloud.)

When a VMM observes a network packet to be delivered to the guest, it sends its proposed virtual time (i.e., in the guest's virtual time, see Section 4) for the delivery of that interrupt to the VMMs on the other machines hosting replicas of the same guest VM. (We stress that these proposals are not visible to the guest VM replicas.) Each VMM generates its proposed delivery time by adding a constant offset Δ_n to the virtual time of the guest VM at its last VM exit. Δ_n must be large enough to ensure that once the three proposals have been collected and the median determined at all three replica VMMs, the chosen median virtual time has not already been passed by any of the guest VMs. The virtual-time offset Δ_n is thus determined using an assumed upper bound on the real time it takes for each VMM to observe the interrupt and to propagate its proposal to the others[4], as well as the maximum allowed difference between the fastest two replicas' virtual times. This difference can be limited by slowing the execution of the fastest replica.

Once the median proposed virtual time for a network interrupt has been determined at a VMM, the VMM simply waits for the first VM exit caused by the guest VM (as in Section 4.2) that occurs at a virtual time at least as large as that median value.[5] The VMM then injects the interrupt prior to the next VM entry of the guest. As with disk accesses and DMA transfers, this interrupt injection also includes copying the data into the address space of the guest so as to prevent the guest VM from polling for the data in advance of the interrupt to create a form of clock (e.g., Hu [1991, Section 4.2.2]).

[4]In distributed computing parlance, we thus assume a *synchronous* system, that is, there are known bounds on processor execution rates and message delivery times.

[5]If the median time determined by a VMM has already passed, then our synchrony assumption was violated by the underlying system. In this case, that VMM's replica has diverged from the others and so must be recovered by, for example, copying the state of another replica.

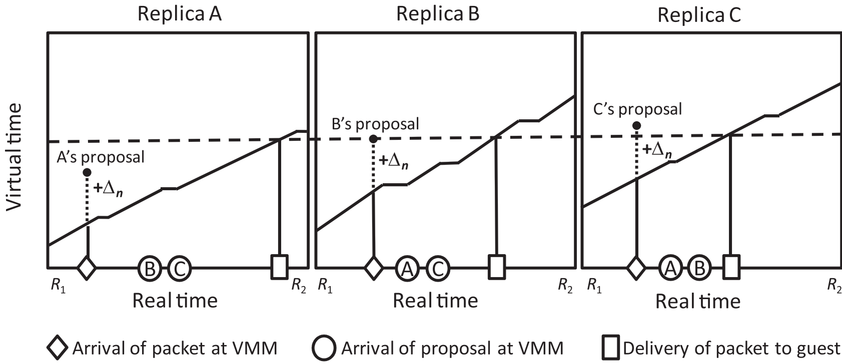


Fig. 2. Delivering a packet to guest VM replicas.

The process of determining the delivery time of a network packet to a guest VM's replicas is pictured in Figure 2. This figure depicts a real-time interval $[R_1, R_2]$ at the three machines at which a guest VM is replicated, showing at each machine: the arrival of a packet at the VMM, the proposal made by each VMM, the arrival of proposals from other replica machines, the selection of the median, and the delivery of the packet to the guest replica. Each stepped diagonal line shows the progression of virtual time at that machine.

5.2. Implementation in Xen

Xen presents to each HVM guest a virtualized platform that resembles a classic PC/server platform with a network card, disk, keyboard, mouse, graphics display, etc. This virtualized platform support is provided by virtual I/O devices (device models) in Dom0, a domain in Xen with special privileges. QEMU (`http://fabrice.bellard.free.fr/qemu`) is used to implement device models. One instance of the device models is run in Dom0 per HVM domain.

5.2.1. Network Card Emulation. In the case of a network card, the device model running in Dom0 receives packets destined for the guest VM. Without StopWatch modification, the device model copies this packet to the guest address space and asserts a virtual network device interrupt via the virtual Programmable Interrupt Controller (vPIC) exposed by the VMM for this guest. HVM guests cannot see real external hardware interrupts, since the VMM controls the platform's interrupt controllers [Intel 2011, Section 29.3.2].

In StopWatch, we modify the network card device model so as to place each packet destined for the guest VM into a buffer hidden from the guest, rather than delivering it to the guest. The device model then reads the current virtual time of the guest (as of the guest's last VM exit), adds Δ_n to this virtual time to create its proposed delivery (virtual) time for this packet, and multicasts this proposal to the other two replicas' device models (step 1 in Figure 3). A memory region shared between Dom0 and the VMM allows device models in Dom0 to read guest virtual time.

Once the network device model receives the two proposals in addition to its own, it takes the median proposal as the delivery time and stores this delivery time in the memory it shares with the VMM. The VMM compares guest virtual time to the delivery time stored in the shared memory upon every guest VM exit caused by guest VM execution. Once guest virtual time has passed the delivery time, the network device model copies the packet into the guest address space (step 2 in Figure 3) and asserts a virtual network interrupt on the vPIC prior to the next VM entry (step 3).

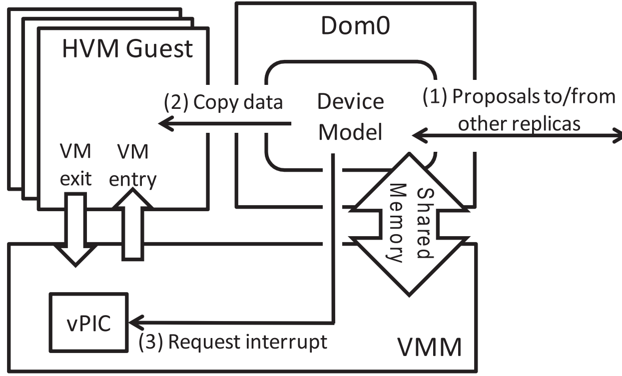
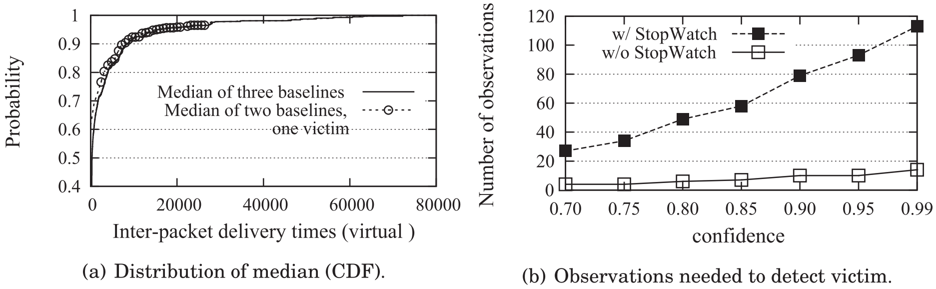


Fig. 3. Emulation of network I/O device in StopWatch.



(a) Distribution of median (CDF).

(b) Observations needed to detect victim.

Fig. 4. Virtual interpacket delivery times to attacker VM replicas with coresident victim (“two baselines, one victim”) and in a run where no replica was coresident with a victim (“three baselines”).

Figure 4(a) shows the CDF of virtual interpacket delivery times to replicas of an attacker VM in an actual run, where one replica is coresident with a victim VM continuously serving a file, in comparison to the virtual delivery times with no victim present. This plot is directly analogous to that in Figure 1(a) but is generated from a real StopWatch run and shows the distribution as a CDF for ease of readability. Figure 4(b) shows the number of observations needed to distinguish the victim and no-victim distributions in Figure 4(a) using a χ-squared test, as a function of the desired confidence. This figure is analogous to Figure 1(b) and confirms that StopWatch strengthens defense against timing attacks by an order of magnitude in this scenario. Again, the absolute number of observations needed to distinguish these distributions is likely quite conservative, owing to numerous practical challenges to gathering these observations [Zhang et al. 2012b].

5.2.2. Disk and DMA Emulation. The emulation of the IDE disk and DMA devices is similar to the preceding network card emulation. StopWatch controls when the disk and DMA device models complete requests and notify the guest. Instead of copying data read to the guest address space, the device model in StopWatch prepares a buffer to receive this data. In addition, rather than asserting an appropriate interrupt via the vPIC to the guest as soon as the data is available, the StopWatch device model reads the current guest virtual time from memory shared with the VMM, adds Δ_d, and stores this value as the interrupt delivery time in the shared memory. Upon the first VM exit caused by guest execution at which the guest virtual time has passed this delivery time, the device model copies the buffered data into the guest address

space and asserts an interrupt on the vPIC. Disk writes are handled similarly in that the interrupt indicating write completion is delivered as dictated by adding Δ_d to the virtual time at which the write was initiated.

6. COLLABORATIVE ATTACKS

The mechanisms described in Section 4–5 intervene on two significant sources of clocks; though VM replicas can measure the progress of one relative to the other, for example, their measurements will be the same and will reflect the median of their timing behaviors. Moreover, by forcing each guest VM to execute (and, in particular, schedule its internal activities) on the basis of virtual time and by synchronizing I/O events across replicas in virtual time, uniprocessor guest VMs execute deterministically, stripping them of the ability to leverage TL and Mem clocks, as well. (More specifically, the progress of TL and Mem clocks are functionally determined by the progress of virtual time and so are not independent of it.) There nevertheless remains the possibility of various collaborative attacks that leverage an attacker VM in conjunction with other attacker components that we discuss next.

6.1. External Collaborators

One possible collaborative attack involves conjoining the attacker VM with a collaborator with which it interacts that is external to the cloud and, in particular, on whose real-time clock we cannot intervene. By interacting with the attacker VM, the external collaborator might attempt to discern information using the real-time behavior of his attacker VM.

Because guest VM replicas will run deterministically, they will output the same network packets in the same order. StopWatch uses this property to interfere with a VM's ability to exfiltrate information on the basis of its real-time behavior as seen by an external observer. StopWatch does so by adopting the median timing across the three guest VM replicas for each output packet. The median is selected at a separate "egress node" that is dedicated for this purpose (c.f., Yin et al. [2002]), analogous to the "ingress node" that replicates every network packet destined to the guest VM to the VM's replicas (see Section 5). Like the ingress node, there need not be only one egress node for the whole cloud.

To implement this scheme in Xen, every packet sent by a guest VM replica is tunneled by the network device model on that machine to the egress node. The egress node forwards an output packet to its destination after receiving the second copy of that packet (i.e., the same packet from two guest VM replicas). Since the second copy of the packet it receives exhibits the median output timing of the three replicas, this strategy ensures that the timing of the output packet sent toward its destination is either the timing of a guest replica not coresident with the victim VM or else a timing that falls between those of guest replicas not coresident with the victim.

An alternative strategy that the external collaborator might take is to send real-time timestamps to his attacker VM, in the hopes of restoring a notion of real time to that VM (that was stripped away as described in Section 4). Again, however, since each packet to the attacker VM is delivered on a schedule dictated by the median progress of the attacker VM replicas (Section 5), those timestamps will reflect only on the behavior of the median replica. As such, it matters little whether the external collaborator sends real-time timestamps to the attacker VM or the attacker VM sends virtual-time timestamps (or events reflecting them) to the external collaborator; either way, the power offered by the external collaborator is the same, namely, relating the median progress of the attacker VM replicas to real time.

6.2. Collaborating Victim-VM Clients

While the type of external collaborator addressed in Section 6.1 interacts with the attacker VM, a more powerful collaborator is one that might additionally interact with the victim VM, for example, as one of its clients. This possibility raises the issue of remote timing attacks (e.g., Brumley and Boneh [2003]) that do not involve coresidence of attacker VMs with victim VMs at all; such attacks are not our concern here, as we are motivated only by *access-driven* attacks.

That said, recent investigations have paired remote timing attacks with access-driven elements: for example, Bates et al. [2012] and Herzberg et al. [2013] developed attacks by which a victim-VM's client could detect the impact of a coresident attacker-VM's communication on the timing of the victim's communication to it, thereby confirming the coresidence of the attacker VM with the victim VM, for example.

While the goal of StopWatch is not to defend against all remote timing attacks, it does mitigate the access-driven elements of attacks, such as those of Bates et al. [2012] and Herzberg et al. [2013]. Specifically, in StopWatch, the observable timing of a victim VM's communication to its clients will be dictated by the median progress of its three replicas (Section 6.1). As shown in Section 3.3, this reveals quantifiably less information to the client than the observable impact of a coresident attacker VM on a (nonreplicated) victim VM would. In particular, an attacker VM could perturb the victim VM's observable communication timings only if it is coresident with the victim VM replica whose progress is the median of the victim's three replicas, and only then constrained above and below according to the other replicas' progress.

The defenses suggested by Herzberg et al. to the attack they investigate include a rate-limiting firewall that interferes with the remote attacker's ability to induce load on VMs hosted in the cloud. Our ingress node (Section 5.1.2) could trivially be adapted to rate-limit inbound traffic as well, as a secondary defense against such attacks.

6.3. Collaborating Attacker VMs

Another possible form of attacker collaboration involves multiple attacker VMs working together to mount access-driven timing attacks. The apparent risks of such collaboration can be seen in the following possibility: replicas of one attacker VM ("VM1") reside on machines A, B, and C; one replica of another attacker VM ("VM2") resides on machine A; and a replica of the victim VM resides on machine C. If VM2 induces significant load on its machines, then this may slow the replica of VM1 on machine A to an extent that marginalizes its impact on median calculations among its replicas' VMMs. The replicas of VM1 would then observe timings influenced by the larger of the replicas on B and C, which may well reflect timings influenced by the victim.

Mounting such an attack, or any collaborative attack involving multiple attacker VMs on one machine, appears to be difficult, however. Just as previously argued that an attacker VM detecting its coresidence with a victim VM is made much harder by StopWatch, one attacker VM detecting coresidence with another using timing covert channels would also be impeded. If the cloud takes measures to avoid disclosing coresidence of one VM with another by other channels, it should be difficult for the attacker to even detect when he is in a position to mount such an attack or to interpret the results of mounting such an attack indiscriminately.

If such attacks are nevertheless feared, they can be made harder still by increasing the number of replicas of each VM. If the number were increased from three to, say, five, then inducing sufficient load to marginalize one attacker replica from its median calculations would not substantially increase the attacker's ability to mount attacks

on a victim. Rather, the attacker would need to marginalize multiple of its replicas, along with accomplishing the requisite setup to do so.

7. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our StopWatch prototype. We present additional implementation details that impact performance in Section 7.1, our experimental setup in Section 7.2, and our tests and their results in Section 7.3–7.4.

7.1. Selected Implementation Details

Our prototype is a modification of Xen version 4.0.2-rc1-pre, amounting to insertions or changes of roughly 1,500 source lines of code (SLOC) in the hypervisor. There were also about 2,000 SLOC insertions and changes to the QEMU device models distributed with that Xen version. In addition to these changes, we incorporated OpenPGM (`http://code.google.com/p/openpgm/`) into the network device model in Dom0. OpenPGM is a high-performance reliable multicast implementation, specifically of the Pragmatic General Multicast (PGM) specification [Speakman et al. 2001]. In PGM, reliable transmission is accomplished by receivers detecting loss and requesting retransmission of lost data. OpenPGM is used in StopWatch for replicating ethernet packets destined to a guest VM to all of that VM's replicas and for communication among the VMMs hosting guest VM replicas. We also extended the network device model on a host to tunnel each ethernet packet emitted from a local VM replica to the appropriate egress node (see Section 6.1) over a persistent TCP connection.

Recall from Section 5 that each VMM proposes (via an OpenPGM multicast) a virtual delivery time for each network interrupt, and the VMMs adopt the median proposal as the actual delivery time. As noted there, each VMM generates its proposal by adding a constant offset Δ_n to the current virtual time of the guest VM. Δ_n must be large enough to ensure that by the time each VMM selects the median, that virtual time has not already passed in the guest VM. However, subject to this constraint, Δ_n should be minimized, since the real time to which Δ_n translates imposes a lower bound on the latency of the interrupt delivery. (Note that because Δ_n is specified in virtual time and virtual time can vary in its relationship to real time, the exact real time to which Δ_n translates can vary during execution.) We selected Δ_n to accommodate timing differences in the arrivals of packets destined to the guest VM at its three replicas' VMMs, the delays for delivering each VMM's proposed virtual delivery time to the others, and the maximum allowed difference in progress between the two fastest guest VM replicas (which StopWatch enforces by slowing the fastest replica, if necessary). For the platform used in our experiments (see Section 7.2) and under diverse networking workloads, we found that a value of Δ_n that typically translates to a real-time delay in the vicinity of 7–12ms sufficed to meet the above criteria. The analogous offset Δ_d for determining the virtual delivery time for disk and DMA interrupts was determined based on the maximum observed disk access times and translates to roughly 8–15ms.

7.2. Experimental Setup

Our "cloud" consisted of three machines with the same hardware configuration: 4 Intel Core2 Quad Q9650 3.00GHz CPUs, 8GB memory, and a 70GB rotating hard drive. Dom0 was configured to run Linux kernel version 2.6.32.25. Each HVM guest had one virtual CPU, 2GB memory, and 16GB disk space. Each guest ran Linux kernel 2.6.32.24 and was configured to use the Programmable Interrupt Controller (PIC) as its interrupt controller and a Programmable Interrupt Timer (PIT) of 250Hz as its clock source. The Advanced Programmable Interrupt Controller (APIC) was disabled. An emulated ATA QEMU disk and a QEMU Realtek RTL-8139/8139C/8139C+ were provided to the guest as its disk and network card. In each of our tests, we installed an

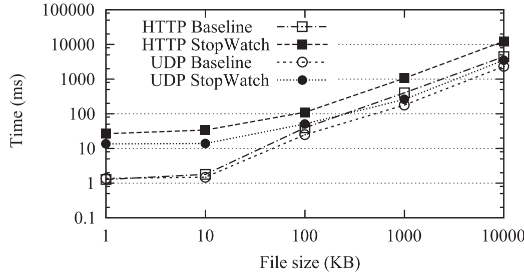


Fig. 5. HTTP and UDP file-retrieval latency.

application (e.g., a Web server or other program) in the guest VM, as will be described later.

After the guest VM was configured, we copied it to our three machines and restored the VM at each. In this way, our three replicas started running from the same state. In addition, we copied the disk file to all three machines to provide identical disk state to the three replicas.

Once the guest VM replicas were started, inbound packets for this guest VM were replicated to all three machines for delivery to their replicas, as discussed in Section 5. These three machines had 100Mb/s ethernet connectivity via a NetGear FS108 switch. They were part of a /24 subnet within the UNC campus network. Broadcast traffic on the network (e.g., ARP requests) was replicated for delivery, as in Section 5. These broadcasts averaged roughly 50–100 packets per second. As such, this background activity was present throughout our experiments and is reflected in our numbers. Since a cloud operator would presumably place the replicas of each VM in close network proximity to one another so as to minimize the networking penalties of coordinating across those machines, we believe that our doing likewise provides a reasonable approximation of the networking costs that StopWatch might encounter in practice.

7.3. Network Services

In this section, we describe tests involving network services deployed on the cloud. In all of our tests, our client that interacted with the cloud-resident service was a Lenovo T400 laptop with a dual-core 2.8GHz CPU and 2GB memory attached to the same /24 subnet as the cloud machines.

7.3.1. File Downloads. Our first experiments tested the performance of file download by the client from a Web server in the cloud. The total times for the client to retrieve files of various sizes over HTTP are shown in Figure 5. This figure shows tests in which our guest VM ran Apache version 2.2.14, and the file retrieval was from a cold start (and so file-system caches were empty). The “HTTP Baseline” curve in Figure 5 shows the average latency for the client to retrieve a file from an unmodified Xen guest VM. The “HTTP StopWatch” curve shows the average cost of file retrieval from our StopWatch implementation. Every average is for ten runs. Note that both axes are log-scale.

Figure 5 shows that for HTTP download, a service running on our current StopWatch prototype loses less than 2.8× in download speed for files of 100KB or larger. Diagnosing this cost reveals that the bottleneck, by an order of magnitude or more, was the network transmission delay (vs. disk access delay) in both the baseline and for StopWatch. Moreover, the performance cost of StopWatch in comparison to the baseline was dominated by the time for delivery of *inbound* packets to the Web-server guest VM, that is, the TCP SYN and ACK messages in the three-way handshake, and then

additional acknowledgments sent by the client. Enforcing a median timing on output packets (Section 6.1) adds modest overhead in comparison.

This combination of insights, namely, the detriment of inbound packets (mostly acknowledgments) to StopWatch file download performance and the fact that these costs so outweigh disk access costs, raises the possibility of recovering file download performance using a transport protocol that minimizes packets inbound to the Web server, for example, using negative acknowledgments or forward error correction. Alternatively, an unreliable transport protocol with no acknowledgments, such as UDP, could be used; transmission reliability could then be enforced at a layer above UDP using negative acknowledgments or forward error correction. Though TCP does not define negative acknowledgments, transport protocols that implement reliability using them are widely available, particularly for *multicast* where positive acknowledgments can lead to "ack implosion." Indeed, recall that the PGM protocol specification [Speakman et al. 2001] and the OpenPGM implementation that we use ensure reliability using negative acknowledgments.

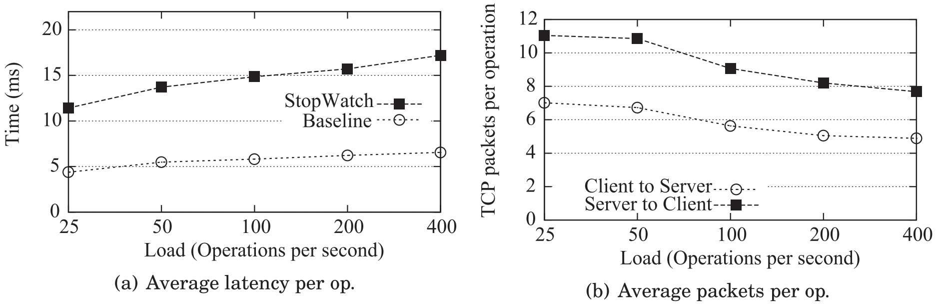
To illustrate this point, in Figure 5, we repeat the experiments using UDP to transfer the file.[6] The "UDP Baseline" curve shows the performance using unmodified Xen; the "UDP StopWatch" curve shows the performance using StopWatch. Not surprisingly, baseline UDP shows performance comparable to (but slightly more efficient than, by less than a factor of two) baseline TCP, but rather than losing an order of magnitude, UDP over StopWatch is competitive with these baseline numbers for files of 100KB or more.

7.3.2. NFS. We also set up a Network File System (NFSv4) server in our guest VM. On our client machine, we installed an NFSv4 client; remotely mounted the filesystem exported by the NFS server; performed file operations manually; and then ran `nfsstat` on the NFS server to print its server-side statistics, including the mix of operations induced by our activity. We then used the `nhfsstone` benchmarking utility to evaluate the performance of the NFS server with and without StopWatch. `nhfsstone` generates an artificial load with a specified mix of NFS operations. The mix of NFS operations used in our tests was the previously extracted mix file.[7] In each test, the client machine ran five processes using the mounted file system, making calls at a constant rate ranging from 25 to 400 per second in total across the five client processes.

The average latency per operation is shown in Figure 6(a). In this figure, the horizontal axis is the rate at which operations were submitted to the server; note that this axis is log-scale. Figure 6(a) suggests that an NFS server over StopWatch incurs a less than 2.7× increase in latency over an NFS server running over unmodified Xen. Since the NFS implementation used TCP, in some sense this is unsurprising in light of the file download results in Figure 5. That said, it is also perhaps surprising that StopWatch's cost increased only roughly logarithmically as a function of the offered rate of operations. This modest growth is in part because StopWatch schedules packets for delivery to guest VM replicas independently—the scheduling of one does not depend on the delivery of a previous one, and so they can be "pipelined"—and because the number of TCP packets from the client to the server actually decreases per operation, on average, as the offered load grows (Figure 6(b)).

[6]We are not advocating UDP for file retrieval generally but rather are simply showing the advantages for StopWatch of a protocol that minimizes client-to-server packets. We did not use OpenPGM in these tests since the website (as the "multicast" originator) would need to initiate the connection to the client; this would have required more substantial modifications. This "directionality" issue is not fundamental to negative acknowledgments, however.

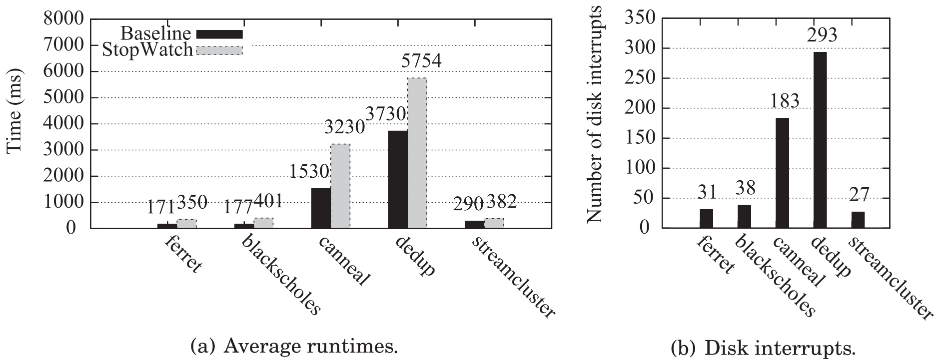
[7]This mix was 11.37% `setattr`, 24.07% `lookup`, 11.92% `write`, 7.93% `getattr`, 32.34% `read`, and 12.37% `create`.



(a) Average latency per op.

(b) Average packets per op.

Fig. 6. Tests of NFS server using `nhfsstone`.



(a) Average runtimes.

(b) Disk interrupts.

Fig. 7. Tests of PARSEC applications.

7.4. Computations

In this section, we evaluate the performance of various computations on StopWatch that may be representative of future cloud workloads. For this purpose, we employ the PARSEC benchmarks [Bienia 2011]. PARSEC is a diverse set of benchmarks that covers a wide range of computations that are likely to become important in the near future (see `http://parsec.cs.princeton.edu/overview.htm`). Here we take PARSEC as representative of future cloud workloads.

We utilized the following five applications from the PARSEC suite (version 2.1), providing each the “native” input designated for it. `ferret` is representative of next-generation search engines for nontext document data types. In our tests, we configured the application for image similarity search. `blackscholes` calculates option pricing with Black-Scholes partial differential equations and is representative of financial analysis applications. `canneal` is representative of engineering applications and uses simulated annealing to optimize routing cost of a chip design. `dedup` represents next-generation backup storage systems characterized by a combination of global and local compression. `streamcluster` is representative of data mining algorithms for online clustering problems. Each of these applications involves various activities, including initial configuration, creating a local directory for results, unpacking input files, performing its computation, and finally cleaning up temporary files.

We ran each benchmark ten times in one guest VM over unmodified Xen, and then ten more times with three guest VM replicas over StopWatch. Figure 7(a) shows the average runtimes of these applications in both cases. In this figure, each application is described by two bars; the black bar on the left shows its performance over unmodified Xen, and the gray bar on the right shows its performance over StopWatch. StopWatch

imposed an overhead of at most 2.3× (for `blackscholes`) to the average running time of the applications. Owing to the dearth of network traffic involved in these applications, the overhead imposed by StopWatch is mostly due to the overhead involved in intervening on disk I/O (see Section 5). As shown in Figure 7(b), there is a direct correlation between the number of disk interrupts to deliver during the application run and the performance penalty (in absolute terms) that StopWatch imposes. If the computers in our experiments used solid-state drives (vs. hard disks), we conjecture that their reduced access times would permit us to shrink Δ_d and so improve the performance of StopWatch for these applications.

8. COMPARISON TO ALTERNATIVES

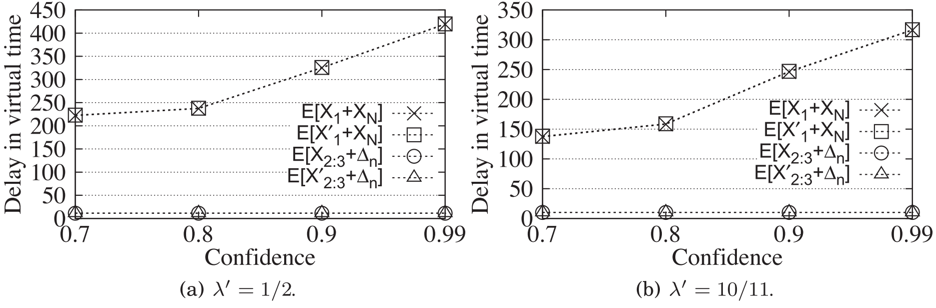
In this section, we pause to compare StopWatch to two alternatives for defending against timing side channels of the form we consider here. The two alternatives we consider, neither of which involves VM replication at all, is (i) overcoming timing side channels by the injection of random noise, and (ii) temporally isolating guest VMs by time slicing each node and running only one guest VM at a time on the node, resetting the machine to as clean a state as possible between each. We discuss these alternatives in Section 8.1 and Section 8.2, respectively.

The purpose of our comparisons is to illustrate certain advantages that StopWatch has over these alternatives, but not to argue that StopWatch is superior to these alternatives in all ways. Indeed, it is appropriate to point out that StopWatch's approach comes with several deployment overheads that these alternatives do not suffer. For example, StopWatch requires VM replication and the placement of each VM's replicas so that the replicas of any VM are coresident with nonoverlapping sets of (replicas of) other VMs, a nontrivial placement constraint discussed further in Section 9. Moreover, for any VM for which networking performance is important, the VM replicas should be placed in close network proximity to one another (as we discussed in Section 7.2). The cloud must additionally provide (not necessarily physically distinct) ingress nodes for replicating inbound traffic to each VM's replicas (Section 5), and egress nodes for hiding timing information in the traffic (the replicas of) each VM sends to others (Section 6.1). Neither of the alternatives discussed next impose such additional requirements.

8.1. Comparison to Uniformly Random Noise

An alternative to StopWatch is simply adding random noise (without replicating VMs) to confound timing attacks. To illustrate advantages that StopWatch's approach has over this alternative, we borrow notation first introduced in Section 3.3: Let X_1 denote a random variable representing the "baseline" timing behavior observed by an attacker VM (replica) in the absence of the victim of interest, and let X'_1 be the random variable as observed by the attacker VM when it is coresident with the victim VM of interest. Again, in StopWatch, the adversary learns information from the difference between (i) the distribution of $X_{2:3}$ for random variables X_1, X_2, X_3 corresponding to attacker VM replicas that are not coresident with a victim VM of interest, and (ii) the distribution of $X'_{2:3}$ for random variables X'_1, X_2, X_3, where X'_1 corresponds to an attacker VM that is coresident with the victim VM of interest. More specifically, in the case where $X_{2:3}$ or $X'_{2:3}$ denotes the logical time of a network interrupt delivery, for example, the adversary observes either $X_{2:3} + \Delta_n$ or $X'_{2:3} + \Delta_n$. (Δ_n is discussed in Section 5.1.2.)

For simplicity, suppose that X_1 and X'_1 are exponentially distributed with rate parameters λ and λ', respectively, as in the example of Figure 1. For the random variable X_{N} representing added noise, assume that X_{N} is drawn uniformly from $[0, b]$ (i.e.,



(a) $\lambda' = 1/2$. (b) $\lambda' = 10/11$.

Fig. 8. Expected delay induced by StopWatch vs. by uniform noise as a function of confidence with which attacker distinguishes the two distributions (coresident victim or not) after the same number of observations; baseline distribution $\text{Exp}(\lambda)$, $\lambda = 1$; victim distribution $\text{Exp}(\lambda')$.

$X_{\text{N}} \sim \text{U}(0, b)$), a common choice to mitigate timing channels (e.g., Hu [1991], Giles and Hajek [2002]).

We calculated expected delay imposed by StopWatch and by adding uniformly distributed noise. To make a fair comparison, we configured both approaches to provide the same strength of defense against timing attacks. Specifically, after calculating the number of observations the attacker requires in the case of StopWatch to distinguish, for a fixed confidence level, the distributions $X_{2:3} + \Delta_n$ and $X'_{2:3} + \Delta_n$ using a χ-squared test, we calculated the minimum b that would give the attacker the same confidence in distinguishing $X_1 + X_{\text{N}}$ and $X'_1 + X_{\text{N}}$ after that number of observations. Figure 8 shows the resulting expected delays in each case.

This figure indicates that StopWatch scales much better as the attacker's required confidence and the distinctiveness of the victim grows (as represented by λ' dropping). The delay of the StopWatch approach is tied most directly to Δ_n, which is added to ensure that the replicas of each VM remain synchronized (see Section 5.1.2); here we calculated it so that $\Pr[\,|X_1 - X'_1| \le \Delta_n] \ge 0.9999$. That is, the probability of a desynchronization at this event is less than 0.0001. Note that $E[X_{2:3} + \Delta_n]$ and $E[X'_{2:3} + \Delta_n]$ are nearly the same in Figure 8, since their difference is how the attacker differentiates the two, and similarly for $E[X_1 + X_{\text{N}}]$ and $E[X'_1 + X_{\text{N}}]$.

8.2. Comparison to Time Slicing

In this section, we compare StopWatch to another alternative, namely, time slicing, to defend against timing attacks. Here, "time slicing" refers to executing each VM (without replication) in isolation for a period of time. When multiple VMs coreside on the same physical machine, they are scheduled to run in a one-at-a-time fashion. Specifically, time is divided into *slices*, and within each time slice, only one VM is allowed to execute, exclusively occupying all physical resources. VMs are scheduled to consume time slices according to a round-robin scheduler (i.e., in turns). In addition, each two consecutive time slices are separated by a cleansing period within which we cleanse shared components in the system to simulate a machine reset. As an example, Figure 9 depicts the execution of three time-sliced VMs running on the same machine.

8.2.1. Design. To make VMs execute in turns, we unpause virtual CPUs (vCPUs) of one VM and leave vCPUs of all the other VMs paused for the duration of a time slice. In this experiment, we designed three sets of cleansing operations, described next and summarized in Table I, to flush shared components in the system with varying degrees of thoroughness. Even our most aggressive cleansing operation falls short of a complete machine reset since there are some shared components (e.g., shared network stack of

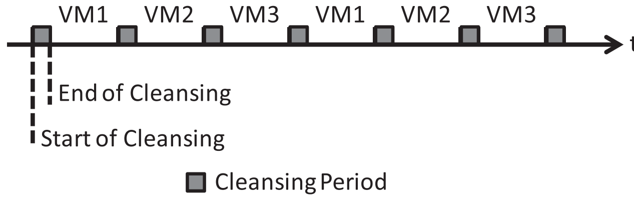


Fig. 9. Time-sliced execution of three VMs.

Table I. Length of Time Slice (*sliceLen*) and of Cleansing (*cleanseLen*)

	sliceLen (s)	*cleanseLen* (s)
Flush-A	0.4	0.001
Flush-B	2	0.2
Flush-C	2.5	0.25

the host machine) with state that could carry information about one VM to another. For each type of cleansing operation described next, we set the length of each time slice (*sliceLen*) to be larger than the length of the cleansing period (*cleanseLen*) by at least one order of magnitude, in an effort to limit the impact of cleansing periods.

Flush-A: CPU Caches and TLB. In the "Flush-A" cleansing operation, we use the `WBINVD` instruction to flush CPU instruction and data caches. `WBINVD` writes back all modified (instruction and data) cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then directs external caches to be invalidated and to write back modified data, though there are no external caches on our machines in this experiment. In our experiment, `WBINVD` is invoked at the beginning of each cleansing period, as depicted in Figure 9.

The TLB (Translation Lookaside Buffer) stores translations between virtual addresses and physical addresses. It gets flushed every time a context switch happens and CR3 register is reloaded. The flushing of the TLB is automatically carried out by the virtualization software we use (Xen).

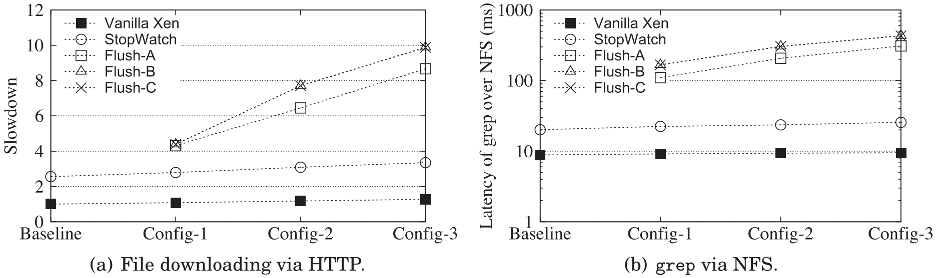
When we choose the length of the cleansing period (*cleanseLen*), we choose a value that is big enough to pause/unpause vCPUs (about 0.8ms in total on our machines) and to complete all flushing operations. In Flush-A, we use *sliceLen* = 0.4s and *cleanseLen* = 0.001s. (In contrast, Xen's CPU schedule quantum is 30ms.)

Flush-B: Flush-A + Disk Page Cache. The disk page cache is a buffer of disk-backed pages kept in main memory (RAM) by the operating system for quicker access. All physical memory that is not directly allocated to applications is usually used by the operating system for the page cache. For a VM running on Xen, the disk device is virtualized and provided by a device model process running in Dom0. QEMU [Bellard 2005] is used to implement such device models which, by default, uses write-through caching for all block devices (see `http://wiki.qemu.org/download/qemu-doc.html`). This means that the page cache of Dom0 will be used to read and write data. To flush the disk page cache, we use a `SYNC` system call followed by writing to /proc/sys/vm/drop_caches. `SYNC` writes all dirty cache pages to the disk, while writing to /proc/sys/vm/drop_caches frees all the page caches for reading. In Flush-B, in addition to the CPU and TLB caches, we flush the disk page cache as well, which takes about 185ms in our system. In this case, we set *sliceLen* = 2.0s and *cleanseLen* = 0.2s.

Flush-C: Flush-B + On-Drive Disk Cache Buffer. The disk cache buffer is the embedded memory in a hard drive acting as a buffer between the rest of the computer

Table II. Configurations

	total hosts	total VMs	VM (replicas) per host		
			vanilla Xen	time slicing	StopWatch
Baseline	3	1	0 or 1	N/A	1
Config-1	13	26	2	2	6
Config-2	19	57	3	3	9
Config-3	25	100	4	4	12



(a) File downloading via HTTP.

(b) `grep` via NFS.

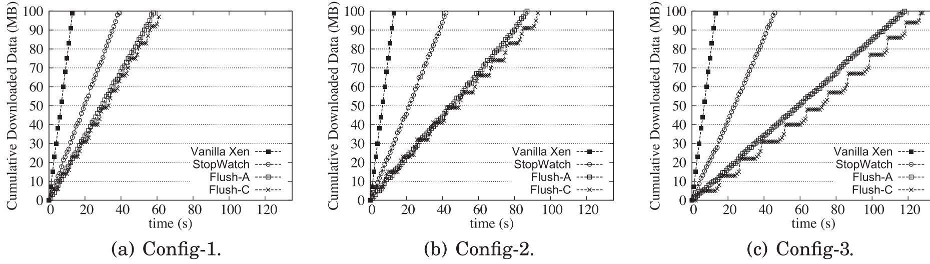
Fig. 10. StopWatch vs. time slicing: comparison of slowdown and delay.

and the physical hard disk platter that is used for storage. We use the utility `hdparm -F`, which takes roughly 25ms, to flush this buffer in addition to operations included in Flush-B. In this case, we set $sliceLen = 2.5$s and $cleanseLen = 0.25$s.

8.2.2. Evaluation. To fairly compare the performance of VMs running under StopWatch and in a time-sliced fashion, we first configure our system carefully so that the same number of VMs are running on the same number of physical machines in both modes. For instance, given 13 machines, if each is time sliced by two VMs, then there are 26 VMs running in total. StopWatch can also support 26 VMs (78 replicas in total) with 13 machines, each of which hosts 6 distinct replicas without violating StopWatch's placement constraints. We have three configurations in this evaluation, shown in Table II, as well as a "Baseline" configuration in which there is at most one VM (replica) per host. In all tests, one "target VM" is serving files via HTTP; half of the other VMs (if any, and rounding up if necessary) with which it is coresident are serving NFS with a workload described next; and the rest are receiving light ICMP traffic (i.e., being `ping`'ed). All VMs in this experiment are uniprocessor VMs. The machines used to support these experiments are as described in Section 7.2.

In Figure 10, we compare the performance of the target VM serving files via HTTP in the time slicing and StopWatch cases. Specifically, the target VM serves a file of size 100MB via HTTP. In these tests, the downloading client was a machine sitting on the same campus network as the nodes hosting these VMs, with a wired connection. The y-axis shows the slowdown factor, which is computed by dividing the time taken to fetch the file from the target VM running in either StopWatch or time-sliced mode by the "vanilla Xen" value for that configuration. Each shown data point is the average over ten such downloads.

To help explain results shown in Figure 10(a), in Figure 11, we show the progress of downloading for various setups. (Flush-B is not shown, since it largely overlays Flush-C.) Even in Flush-A, the download speed suffers both from frequent context switches among VMs and from CPU cache flushing. While in Flush-C, which has longer time slices, the download speed roughly recovers within one slice from a cleaned cache, the slice ends shortly thereafter. And also due to the longer time slices, stepped effects become more obvious in Flush-C.



(a) Config-1. (b) Config-2. (c) Config-3.

Fig. 11. Progress of file download via HTTP.

Finally, in Figure 10(b), we confirm these effects by measuring the latency of highly interactive NFS operations. An NFS server was set up in the target VM, and the client remotely mounted the exported partition and then launched `grep` operations, trying to find a target string in a 32B file. `grep` operations were conducted with a frequency of 10 ops/s, and the average latency to perform 200 `grep` operations is reported. In this experiment, the effects observed in the HTTP case manifest themselves as paused `grep` commands owing to the NFS server not being scheduled yet and so being unable to respond.

8.3. Discussion

The previous analyses are not meant to conclude that StopWatch will always provide superior performance to adding random noise or time slicing hosts, nor do we believe that is the case. For example, machines with few physical cores and a compute-intensive batch workload would almost certainly perform better with time slicing than it would with StopWatch, since StopWatch would triplicate the computations on machines allowing minimal concurrency. That said, the analyses do illustrate ways in which StopWatch could outperform these alternative designs, while providing an arguably more holistic defense against timing channels than either of them.

9. REPLICA PLACEMENT IN THE CLOUD

StopWatch requires that the three replicas of each guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. This constrains how a cloud operator places guest VM replicas on its machines. In this section, we clarify the significance of these placement constraints in terms of the provider's ability to best utilize its infrastructure. After all, if under these constraints, the provider were able to simultaneously run a number of guest VMs that scales, say, only linearly in the number of cloud nodes, then the provider should forgo StopWatch and simply run each guest VM (nonreplicated) in isolation on a separate node. Here we show that the cloud operator is not limited to such poor utilization of its machines.

If the cloud has n machines, then consider the complete, undirected graph (clique) K_n on n vertices, one per machine. For every guest VM, the placement of its three replicas forms a *triangle* in K_n consisting of the vertices for the machines on which the replicas are placed and the edges between those vertices. The placement constraints of StopWatch can be expressed by requiring that the triangles representing VM replica placements be pairwise *edge-disjoint*. As such, the number of guest VMs that can simultaneously be run on a cloud of n machines is the same as the number of edge-disjoint triangles that can be *packed* into K_n. A corollary of a result due to Horsley [2011, Thm. 1.1] is as follows.

THEOREM 3. *A maximum packing of K_n with pairwise edge-disjoint triangles has exactly k triangles, where (i) if n is odd, then k is the largest integer such that* $3k \le \binom{n}{2}$ *and* $\binom{n}{2} - 3k \notin \{1, 2\}$*; and (ii) if n is even, then k is the largest integer such that* $3k \le \binom{n}{2} - \frac{n}{2}$.

So, a cloud of n machines using StopWatch can simultaneously execute $k = \Theta(n^2)$ guest VMs. The existence of such a placement, however, does not guarantee an efficient algorithm to find it. Moreover, this theorem ignores machine capacities. Next we address both of these shortcomings.

Under the constraints of StopWatch, one node in a cloud of n nodes can simultaneously execute up to $\frac{n-1}{2}$ guest VMs, since the other replicas of the guest VMs that it executes (two per VM) must occupy distinct nodes. If each node has resources to simultaneously execute $c \le \frac{n-1}{2}$ guest VMs, then the following theorem provides for an algorithm to efficiently place them subject to the per-machine capacity constraint c. Its proof can be found in the Online Appendix available in the ACM Digital Library.

THEOREM 4. *Let* $n \equiv 3 \bmod 6$ *and* $c \le \frac{n-1}{2}$. *If* $c \equiv 0$ *or* $1 \bmod 3$, *then there is an efficient algorithm to place* $k \le \frac{1}{3}cn$ *guest VMs. If* $c \equiv 2 \bmod 3$, *then there is an efficient algorithm to place* $k \le \frac{1}{3}(c-1)n + \frac{n-3}{6}$ *guest VMs.*

A limitation of Theorem 4 is that it provides an efficient algorithm to place $\Theta(cn)$ VMs only in the case that all VMs consume one unit of machine capacity. In this sense, the theorem is simplistic, since VMs submitted to clouds frequently have different demands for some resources. For example, if the capacity c represents physical memory, then different VMs may have different memory demands. The following theorem provides for an efficient placement of VMs even in this case, and is proved in the Online Appendix.

THEOREM 5. *Let* $n = 6v + 3$, *and* $2v + 1 = 3^q$ *for some* $q \in \mathbb{N}$. *Suppose that each machine has capacity* $c \le \frac{n-1}{2}$ *and each VM guest has a constant associated demand on that capacity of at most* d_{max}. *There is an efficient algorithm to place* $\Theta(\frac{c}{d_{max}}n)$ *VM guests.*

10. CONCLUSION

We proposed a new method for addressing timing side channels in IaaS compute clouds that employs three-way replication of guest VMs and placement of these VM replicas so that they are coresident with nonoverlapping sets of (replicas of) other VMs. By permitting these replicas to observe only virtual (vs. real) time and the median timing of I/O events across the three replicas, we suppress their ability to glean information from a victim VM with which one is coresident. We described an implementation of this technique in Xen, yielding a system called StopWatch, and we evaluated the performance of StopWatch on a variety of workloads. Though the performance cost for our current prototype ranges up to 2.8× for networking applications, we used our evaluation to identify the sources of costs and alternative application designs (e.g., reliable transmission using negative acknowledgments, to support serving files) that can enhance performance considerably. We also extended this evaluation to demonstrate ways in which StopWatch can provide better performance than alternatives that leverage commodity hardware, namely, adding random noise to observable event timings and eliminating concurrent VM execution (time slicing). We showed that clouds with

n machines capable of each running $c \leq \frac{n-1}{2}$ guest VMs simultaneously can efficiently schedule $\Theta(cn)$ guest VMs under the constraints of StopWatch, or $\Theta(\frac{cn}{d_{max}})$ guest VMs if each guest VM makes demands on the per-machine capacity c of at most d_{max}. These results represent a clear improvement over the alternative of running each guest VMs on its own machine. We envision StopWatch as a basis for a high-security cloud, for example, suitable for military, intelligence, or financial communities with high assurance needs.

An important topic for future work is extending StopWatch to support multiprocessor guest VMs. As discussed in Section 2, previous research on deterministic scheduling (e.g., Devietti et al. [2010]) should provide a basis for extending our current StopWatch prototype. A second direction for improvement is that we have implicitly assumed in our StopWatch implementation (and in many of our descriptions in this article) that the replicas of each VM are placed on a set of homogeneous nodes. Expanding our implementation to heterogeneous nodes poses additional challenges that we hope to address in future work. Finally, as discussed in Section 8, StopWatch introduces various deployment considerations (e.g., ingress and egress nodes, and VM replication with replica placement constraints) that some competing alternatives do not. An additional topic for future work is evaluating the operational costs of these requirements in practice and possibly improving the StopWatch design to reduce these burdens on cloud operators while retaining the advantages it offers.

REFERENCES

J. Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. 40–53.

A. Askarov, A. C. Myers, and D. Zhang. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. 520–538.

A. Aviram, S.-C. Weng, S. Hu, and B. Ford. 2010. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*.

C. Basile, Z. Kalbarczyk, and R. K. Iyer. 2006. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst. 17*, 5, 448–465.

A. Bates, B. Mood, J. Fletcher, H. Pruse, M. Valafar, and K. Butler. 2012. Detecting co-residency with active fraffic analysis techniques. In *Proceedings of the ACM Workshop on Cloud Computing Security*. 1–12.

F. Bellard. 2005. QEMU, a fast and protable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.

C. Bienia. 2011. Benchmarking Modern Multiprocessors. Ph.D. Dissertation. Princeton University.

A. Borg, W. Blau,W. Graetsch, F. Herrmann, and W. Oberle. 1989. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 7*, 1, 1–24.

T. C. Bressoud and F. B. Schneider. 1996. Hypervisor-based fault-tolerance. *ACM Trans. Comput. Syst. 14*, 1, 80–107.

D. Brumley and D. Boneh. 2003. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*. 1–14.

B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*.

B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 161–174.

J. Devietti, B. Lucia, L. Ceze, and M. Oskin. 2010. DMP: Deterministic shared memory multiprocessing. *IEEE Micro 30*, 41–49.

E. Deza and M. Deza. 2006. *Dictionary of Distances*. Elsevier.

G. W. Dunlap, D. G. Lucchetti, P. M. Chen, and M. A. Fetterman. 2008. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM Conference on Virtual Execution Environments*. 121–130.

D. Gao, M. K. Reiter, and D. Song. 2005. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium Recent Advances in Intrusion Detection*. 63–81.

D. Gao, M. K. Reiter, and D. Song. 2009. Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance. *IEEE Trans. Depend. Secure Comput. 6*, 2, 96–110.

J. Giles and B. Hajek. 2002. An information-theoretic and game-theoretic study of timing channels. *IEEE Trans. Inf. Theory 48*, 9.

M. Güngör, Y. Bulut, and S. Çalık. 2009. Distributions of order statistics. *Appl. Math. Sci. 3*, 16, 795–802.

A. Haeberlen, B. C. Pierce, and A. Narayan. 2011. Differential privacy under fire. In *Proceedings of the 20th USENIX Security Symposium*.

A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl. 2013. Cloudoscopy: Services discovery and topology mapping. In *Proceedings of the ACM Workshop on Cloud Computing Security*. 113–122.

D. Horsley. 2011. Maximum packing of the complete graph with uniform length cycles. *J. Graph Theory 68*, 1, 1–7.

W.-M. Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Symposium on Security and Privacy*. 8–20.

Intel. 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

M. H. Kang and I. S.Moskowitz. 1993. A pump for rapid, reliable, secure communication. In *Proceedings of the ACM Conference on Computer and Communications Security*. 119–129.

T. Karagiannis, M. Molle, M. Faloutsos, and A. Broido. 2004. A nonstationary Poisson view of Internet traffic. In *Proceedings of the INFOCOM*. 1558–1569.

T. Kim, M. Peinado, and G. Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*.

C. C. Lindner and C. A. Rodger. 2008. *Design Theory*. CRC Press, Chapter 1.

P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. 1999. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. 263–273.

A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson. 2008. Security through redundant data diversity. In *Proceeding of the 38th IEEE/IFPF International Conference on Dependable Systems and Networks*.

G. Popek and C. Kline. 1974. Verifiable secure operating system software. In *Proceedings of the AFIPS National Computer Conference*. 145–151.

H. Raj, R. Nathuji, A. Singh, and P. England. 2009. Resource management for isolation enhanced cloud services. In *Proceedings of the ACM Workshop on Cloud Computing Security*. 77–84.

T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 199–212.

F. B. Schneider. 1987. Understanding Protocols for Byzantine Clock Synchronization. Technical Report 87-859, Department of Computer Science, Cornell University.

F. B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surveys 22*, 4.

T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekara, and L. Vicisano. 2001. PGM reliable transport protocol specification. Request for Comments 3208, Internet Engineering Task Force.

E. Tromer, D. A. Osvik, and A. Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *J. Cryptol. 23*, 1, 37–71.

R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. 2005. Intel virtualization technology. *IEEE Computer 38*, 3, 48–56.

B. C. Vattikonda, S. Das, and H. Shacham. 2011. Eliminating fine grained timers in Xen. In *Proceedings of the ACM Cloud Computing Security Workshop*.

VMWare. 2010. *Timekeeping in VMware Virtual Machines*. VMWare Inc.

J. C. Wray. 1991. An analysis of covert timing channels. In *Proceedings of the IEEE Symposium on Security and Privacy*. 2–7.

M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. 2007. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Workshop on Modeling, Benchmarking and Simulation*.

J. Yin, A. Venkataramani, J.-P. Martin, L. Alvisi, and M. Dahlin. 2002. Byzantine fault-tolerant confidentiality. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*.

S. Zdancewic and A. C. Myers. 2003. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*. 29–43.

D. Zhang, A. Askarov, and A. C. Myers. 2011. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*.

D. Zhang, A. Askarov, and A. C. Myers. 2012a. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM Conference on Programming Language Design and Implementation*.

Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. 2012b. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*.

Received January 2014; revised June 2014; accepted September 2014