StopWatch: Toward "Differentially Private" Timing for Cloud Executions

Peng Li
Department of Computer Science
University of North Carolina
Email: pengli@cs.unc.edu

Debin Gao
School of Information Systems
Singapore Management University
Email: dbgao@smu.edu.sg

Michael K. Reiter
Department of Computer Science
University of North Carolina
Email: reiter@cs.unc.edu

***Abstract*—This paper describes *StopWatch*, a system that defends against timing-based side-channel attacks that arise from coresidency of victims and attackers in infrastructure-as-a-service cloud environments. *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. Then, *StopWatch* uses the timing behaviors of a VM's replicas collectively to determine the timing of events observed by each one or by an external observer, so that observable timing behaviors could have been observed in the absence of any other individual, coresident VM. In this respect, *StopWatch* implements a form of "differentially private" timing behavior in a cloud. We detail the design and implementation of *StopWatch* in Xen, evaluate the factors that influence its performance, and address the problem of placing VM replicas in a cloud under the constraints of *StopWatch* so as to still enable adequate cloud utilization.**

***Keywords*-timing attacks; side channels; replication; clouds**

I. Introduction

Implicit timing-based information flows potentially threaten the use of clouds for highly sensitive computations. In an "infrastructure as a service" (IaaS) cloud, such an attack would be mounted by an attacker submitting an attacker virtual machine (VM) to the cloud that times the duration between events that it can observe, in order to make inferences about a *victim* VM with which it is running simultaneously on the same host but otherwise cannot access. Such attacks 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.

A known defense against timing-based covert channels and side channels is to perturb (e.g., through randomization or coarsening) clocks that are visible to VMs, making it more difficult for the attacker VM to measure the duration between events and so to receive the signals (e.g., [1]). While this technique slows (but does not entirely defeat) timing-based channels, the protection it offers can be difficult or impossible to quantify when applied heuristically.

This state of affairs is reminiscent of another subdomain of the security field, namely inference control in the release of datasets of sensitive information (e.g., health records) about people. Perturbation of query results (randomization and coarsening again being notable examples) has been a staple of that subdomain for decades (e.g., [2]). Recently, a formal underpinning to guide its application has started to gain momentum, namely *differential privacy*. "Achieving differential privacy revolves around hiding the presence or absence of a single individual" [3] in a dataset.

In this paper we adapt this intuitive goal of differential privacy — i.e., that the adversary cannot discern from his observations whether a person is represented in a dataset — to the entirely different domain of timing attacks in the cloud. More specifically, we develop a system called *StopWatch* that perturbs timing signals available to an attacker VM so that these signals could have been observed in the absence of the victim, irrespective of the distinctiveness of the victim within the cloud workload. Due to our different domain, however, the methods employed in *StopWatch* to achieve this property are wholly different than randomizing query responses. Rather, *StopWatch* perturbs timings observed by the attacker VM to "match" those of a *replica* attacker VM that is *not* coresident with the victim.

Since *StopWatch* cannot identify attackers and victims, 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. Moreover, two replicas is not enough: one might be coresident with its victim, and by symmetry, the timings it observes would necessarily influence the timings imposed on the pair. *StopWatch* thus uses three replicas that coreside with nonoverlapping sets of (replicas of) other VMs and imposes the timing of the "median" of the three on all replicas. Even if the median timing of an event is that observed by an attacker replica that is coresident with a victim replica, attacker replicas that do not coreside with the victim observed timings both below and above the median.[1]

We detail the implementation of *StopWatch* in Xen,

[1]The median can be viewed as "microaggregating" the timings to confound inferences from them (c.f., [4]–[6]). This analogy suggests the possibility of using other microaggregation functions, as well, of which there are many [7]. We do not pursue that possibility here.

specifically to enforce this "median" behavior on all real-time clocks and "clocks" available via the I/O subsystem (e.g., disk and network interrupts). In doing so, *StopWatch* interferes with all timing side-channel attacks commonly used in the research literature, owing to the normal use of real-time as a reference clock in those exploits. (Timing attacks that do not use real-time clocks should generally be more fragile due to unpredictable influences on other reference clocks.) Moreover, for uniprocessor VMs, *StopWatch* enforces deterministic execution across all of a 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 this 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 extensively evaluate the performance of our *StopWatch* prototype for supporting web service (file downloads), network file systems, and various types of computational tasks. Our analysis reveals the primary factors that influence the performance penalties that *StopWatch* imposes; e.g., inbound packets to a *StopWatch*-supported network service incur much larger overheads than outbound ones. This enables us to identify adaptations to a service that can vastly increase its performance when run over *StopWatch*. For example, we show that reliable transport using negative acknowledgements (or unreliable transport with no acknowledgements, as in UDP) versus positive ones (as in TCP) can dramatically improve file download latencies in the common case of few losses, even to the extent of making file download over *StopWatch* competitive with file download over unmodified Xen. For computational benchmark programs, we find that the overheads induced by *StopWatch* are directly correlated with the amount of disk I/O they perform.

We also analyze a utilization question that would be faced by cloud operators if they were to make use of *StopWatch*, namely how many guest VMs can be simultaneously executed on an infrastructure of n machines under the constraint that the three replicas for each guest VM coreside with nonoverlapping sets of (replicas of) other VMs. We relate this question to a graph-theoretic problem studied in computational biology and find that $\Theta(n^2)$ guest VMs can be simultaneously executed. We also identify practical algorithms for placing replicas to achieve this bound.

To summarize, our contributions are as follows:

- We introduce a novel approach for defending against 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 timing of any event across the three guest VM replicas is then imposed on all of its replicas to approximate a "differentially private" observation of event timings.
- 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 identify algorithmic results from graph theory and computational biology that resolve the problem of how to place replicas under the constraints of *StopWatch* to maximally utilize a cloud infrastructure.

The rest of this paper is structured as follows. We describe related work in §II. We provide an overview of the design of *StopWatch* in §III. We then detail how we address classes of internal "clocks" used in timing attacks in §IV and §V, and then turn to timing as seen by an outside observer in §VI. We evaluate performance of our *StopWatch* prototype in §VII. §VIII treats the replica placement problem that would be faced by cloud operators using *StopWatch*. We discuss future work in §IX and conclude in §X.

II. Related Work

A. *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., [8], [9]), 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., [10], [11]), but we seek an approach that applies to general computations.

Askarov et al. [12] 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., [1], [8], [9]) and external (e.g., [12]–[16]) timing channels have received significant attention individually, though to our knowledge, *StopWatch* is novel in addressing 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, as well. *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 timings from the three replicas. That is, each internal and external timing observation is of either an attacker VM replica that is not coresident with a victim VM replica or else lies between timings of such replicas.

B. 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 [17], [18] to mask Byzantine faults [19] 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 §III and §VI). 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., [20]–[25]), but not all (e.g., [26]), 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 and, eventually, to leak it 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 developed for Byzantine fault-tolerant clock synchronization (e.g., see [27, §5.2]).

Aside from replication for fault tolerance, replication has been explored to detect server penetration [28]–[31]. 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., [32], [33]) 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 an attacker VM trying to leak information about a victim with which it coresides. That said, the state-of-the-art in this domain (e.g., [33]) addresses multiprocessor VM execution, which our present implementation of *StopWatch* does not. We expect that *StopWatch* could be extended to support multiprocessor execution with techniques developed for replay of multiprocessor VMs, and we plan to investigate this in future research. Mechanisms for enforcing deterministic execution of parallel computations through modifications at user level (e.g., [34], [35]) or the operating system (e.g., [36]) are less relevant to our goals, as they are not easily utilized by an IaaS cloud provider that accepts arbitrary VMs for execution.

III. 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., [37]), 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).

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 are effective in preventing 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. We assume that the cloud's virtualization software (in our case, Xen and our extensions thereof) is not compromised. We allow that the attacker might determine if its VM (or one of its replicas) is colocated with a victim VM replica, though we assume that the attacker VM would generally be unable to assemble a complete inventory of VMs with which it (or more specifically all of its replicas, see below) are coresident.

According to Wray [38], 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, in principle a clock can be any sequence of observable events. With this general definition of a "clock," a timing attack simply involves measuring one clock using another. Wray identified four possible clock sources in conventional computers [38]:

- TL: the "CPU instruction-cycle clock", i.e., 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); and
- RT: real-time clocks provided by the hardware platform (e.g., time-of-day registers).

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 §II-B, extension to multiprocessor VMs remains a topic of future work.) IO and RT (especially RT) clocks are an ingredient in every documented 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. As such, intervening on these clocks is of paramount importance. Moreover, the manner in which we do so implies that the scheduler in a uniprocessor guest VM will behave deterministically, which interferes with any attempts to use TL or Mem clocks.

More specifically, to interfere with IO clocks, *StopWatch* replicates each attacker VM (or, really, 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 event is made available to each replica, the median timing value of the three is adopted. The median timing is either observed at an attacker replica that is not coresident with a victim VM replica or else falls between timings of the two replicas that are not coresident with the victim VM. In the latter case, assuming that the two replicas not coresident with the victim do not themselves offer timings that are extreme, this leaves little latitude for the median timing to convey useful information to the attacker. *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, an idea due to Popek and Kline [39]. Optionally, *StopWatch* adjusts the progression of virtual time periodically using the median real time of the three replicas at fixed points in their executions, thereby roughly synchronizing their views of real time with actual real time to a degree. As discussed in §II, we view *StopWatch*'s use of medians in addressing IO and RT clocks as one of our primary conceptual innovations.

A side effect of how *StopWatch* addresses IO and RT clocks is that it enforces deterministic execution of uniprocessor attacker VM replicas, additionally 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 still glean information from the real-time duration between the arrival of packets that the attacker VM sends. To interfere with this timing channel, we again apply the principle of emitting packets to an external observer with timing dictated by the median timing of the three VM replicas.

The general strategy of permitting only the median timing of each event to be observed, which pervades our design, ensures that an observer sees a timing that is dictated by a replica that is not coresident with the victim VM or else that is bound above and below by such timings. (And, of course, the observer does not learn which replica's timing was adopted.) Intuitively, the protection offered by the latter case derives from the limited latitude for modulating clocks that these bounds allow, but this protection hinges on an additional assumption that these bounds are not too loose. As an extreme case, suppose that one attacker VM replica runs alone, one is coresident with a massive workload that nearly starves it, and the third is coresident with only the victim VM. In this case, the third's timings will likely be the median that is observed, and the presumably large gap between the timings of the first and second leaves plenty of latitude for the third to observe modulations of clocks induced by the victim. As such, VM replicas must be provided the resources needed to support reasonably tight bounds on execution rates of VM replicas, an assumption which is also needed for functional correctness in our design (see §V).

IV. RT CLOCKS

Real-time clocks provide reliable and intuitive reference clocks for measuring the timings of other event sequences. 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.

A. 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 [39]. That is, after the VM executes $instr$ instructions, the virtual time observed from within the VM is

$$virt(instr) \leftarrow slope \times instr + start \qquad (1)$$

To determine $start$ at the beginning of VM replica execution, the VMMs on the machines 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.

Optionally, the VMMs can adjust $start$ and $slope$ periodically, e.g., after the VM replicas execute an "epoch" of I instructions, in an effort to coarsely synchronize $virt$ and real time. For example, after the k-th epoch, each VMM hosting a replica can send to the others the duration D_k over which it 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 reported by that same machine and reset

$$\begin{aligned} start_{k+1} &\leftarrow virt_k(I) \\ slope_{k+1} &\leftarrow \arg \min_{v \in [\ell, u]} \left| \frac{R^*_k - virt_k(I) + D^*_k}{I} - v \right| \end{aligned}$$

for a preconfigured constant range $[\ell, u]$, to yield the formula for $virt_{k+1}$.[2] 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.

B. Implementation in Xen

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

1) Timer interrupts: Operating systems typically measure the passage of time by counting timer interrupts; i.e., the operating system sets up a hardware device to interrupt periodically at a known rate, such as 100 times per second [40]. 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 (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 $instr$, i.e., 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 accurately track the branch count for a guest, *StopWatch* implements a *virtualized* branch counter for each guest. The VM Control Structure (VMCS), through which guest execution is controlled, provides a heap area to save and restore model-specific register (MSR) values such as the hardware branch counter during VM *exits* and *entries*, respectively (described below). Using this mechanism, *StopWatch* tracks the branch count for each guest and uses it for $instr$ in (1).

A remaining question is precisely 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 [41]. While the VMM (e.g., the Xen hypervisor) 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 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 provide 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 will observe the same timer interrupts at precisely 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 [42, §29.3.2]. For VM exits caused by guest VM execution, the VMM injects any needed timer interrupts on the next VM entry.

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 using a constant parameter. *StopWatch* replaces this use of a real-time clock with the guest's virtual clock (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. As in the case of the `rdtsc` call, *StopWatch* responds to requests to read the RTC using the guest's virtual time in place of real time.

3) Reading counters: Other sources from which the guest can observe real time are various hardware counters, e.g., the PIT counter, which repeatedly counts down to zero (at a pace dictated by real time) starting from a constant value. A guest VM can issue a port I/O request to read the current PIT counter value. These counters, too, are already virtualized in modern VMMs such as VMWare (see [40]) and Xen. In Xen, these return values are calculated using a real-time clock, and so *StopWatch* instead uses the guest virtual time (1) in place of real time, as before.

[2] 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 $(R^*_k - virt_k(I) + D^*_k)/I$.

V. 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 §V-A, and then in §V-B we describe our implementation of this strategy in *StopWatch*.

A. Strategy

Recall that *StopWatch* replicates each guest VM threefold and controls the timing of IO clocks seen at each replica by taking a median of proposed timings from the three VMMs hosting them. In order to solicit proposed timings from the three, it is necessary, of course, that the VMMs hosting the three replicas all observe each event. As such, in *StopWatch* we replicate every network packet to all three computers hosting replicas of the VM for which the packet is intended. This is done using a logically separate "ingress node" that we envision residing on a dedicated computer in the cloud. This replication at runtime is not necessary for disk blocks retrieved from the local disk; the replication of each 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. For similar reasons, DMA transfers do not require replication in *StopWatch*.

When a VMM observes a data transfer that should eventually cause an IO interrupt to be delivered to the guest, it sends its proposed virtual time — i.e., in the guest's virtual time, see §IV — for the delivery of that interrupt to the VMMs on the other machines hosting replicas of the same guest VM. It generates its proposed delivery time by adding a constant offset Δ to the virtual time of the guest VM at its last VM exit. Δ 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 these VMMs. Δ 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. In distributed computing parlance, we thus assume that the system is *synchronous*, i.e., that there exist known bounds on processor execution rates and message delivery times. The synchronous model has been widely used to develop reliable distributed protocols (e.g., [19], [43]).

Once the median proposed virtual time for an IO interrupt has been determined at a VMM, the VMM simply waits for the first VM exit caused by the guest VM (as in §IV-B) that occurs at a virtual time at least as large as that median value. 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 [1, §4.2.2]).

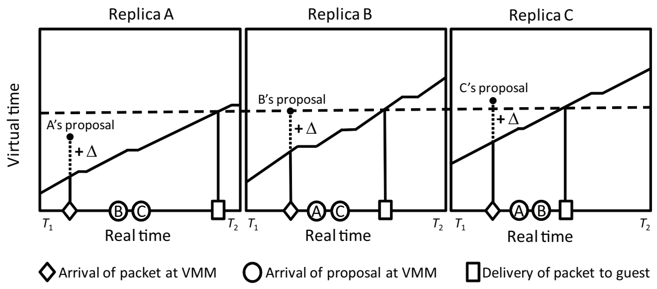


Figure 1. The steps involved in delivering a packet to guest VM replicas.

The process of determining the delivery time of a network packet to guest VMs replicas is pictured in Fig. 1. This figure depicts a real-time interval $[T_1, T_2]$ 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.

B. 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. (See Fig. 2.)

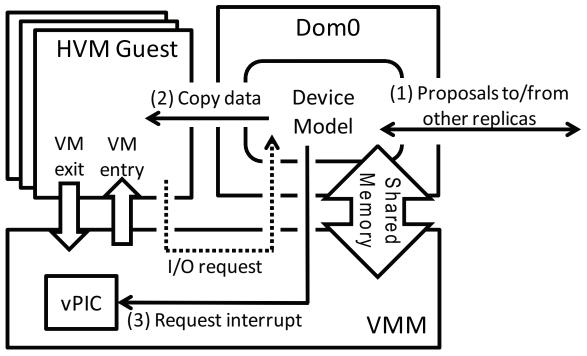


Figure 2. Emulation of I/O devices in *StopWatch*. "I/O request" present only for disk I/O.

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 HVM for this guest. HVM guests cannot see real external hardware interrupts since the VMM controls the platform's interrupt controllers [42, §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 from a shared memory the current virtual time of the guest (as of the guest's last VM exit), adds Δ to this virtual time to create its proposed delivery (virtual) time for this packet, and multicasts this proposal to the other two replicas (step 1 in Fig 2). A memory region shared between Dom0 and the VMM allows device models in Dom0 to read guest virtual time, which is computed and updated on every VM exit by the VMM.

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 (1) 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 Fig. 2) and asserts a virtual network interrupt on the vPIC prior to the next VM entry (step 3).

2) Disk emulation: The emulation of the IDE disk device is performed similarly to the network card emulation above. *StopWatch* controls when the disk device model completes disk I/O requests and notifies the guest. Instead of copying data read from virtual disk sectors to the guest address space, the device model in *StopWatch* prepares a buffer to receive this data. In addition, rather than asserting a virtual IDE disk interrupt via the vPIC to the guest as soon as the data is available from disk, the *StopWatch* device model reads the current guest virtual time from memory shared with the VMM, adds Δ, and then multicasts its proposal for this interrupt to the other replicas. Once it has collected all three proposals, the disk device model stores the median 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 disk device model copies the buffered data into the guest address space and asserts a virtual disk interrupt on the vPIC. Disk writes are handled similarly, in that the interrupt indicating write completion is delivered as dictated by the median of the proposals from the replica's device models.

3) DMA emulation: The DMA device is also emulated in Xen. Once activated, the DMA device model continues data transfer into the guest address space on its own; when the data transfer is completed, the DMA device model issues an interrupt request. Since guest virtual time can be seen by device models running in Dom0 (via the memory shared with the VMM), we can modify the DMA device model to transfer data at rate tied to guest virtual time, not real time, so that a guest VM cannot create an IO clock independent of RT clocks by polling its memory for DMA data. We can also postpone the virtual DMA interrupt until our guests have passed the median proposed virtual time for this interrupt, as in the network and disk cases above. While not yet having completed an implementation of these modifications to the DMA device model, we see no obstacles to doing so and plan to complete this work for the final version of the paper. For the evaluation results in §VII, we disable DMA.

C. Example execution

In this section we illustrate the mechanisms described in §V-A–V-B by showing the progression of delivery times proposed by the three replicas of a guest VM for a series of inbound packets and, in particular, how *StopWatch* interferes with a potential timing side channel. In the execution pictured in Fig. 3, guest VM replicas "B" and "C" ran alone on their respective machines, whereas replica "A" was coresident with another VM (the "victim") that sent a continuous stream of traffic during the run. Due to resource contention on the machine hosting replica A, this replica executed slightly behind replicas B and C in real time; this is shown by A's proposals for the delivery of inbound packets occurring at *earlier* virtual times. Since the median delivery time was selected for each inbound packet, replica A's proposals were not selected as the delivery times. Rather, the delivery time selected for each inbound packet varied between B's and C's over time. In this way, the activity of the victim coresident with A is hidden from the replicas.

VI. External Observers

The mechanisms described in §IV–V 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) 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 that an external observer, on whose real-time clock we cannot intervene, could discern information on the basis of the real-time behavior of his attacker VM. In this section we describe our approach to addressing this form of timing channel.

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 physically separate "egress node" that is dedicated for

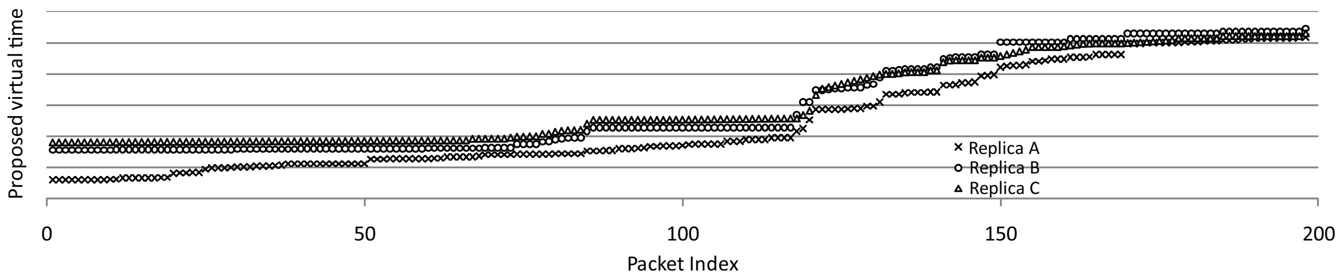


Figure 3. Virtual delivery times proposed by three replicas of a guest VM per inbound packet in an actual execution. Replica A was coresident with a "victim" VM inducing resource contention with replica A, causing it to run behind replicas B and C.

this purpose, analogous to the "ingress node" that replicates every network packet destined to the guest VM to each of the VM's replicas (see §V).

To implement this in the context of Xen, every packet sent by a guest VM replica is forwarded by the network device model on that machine to the egress node over TCP. The egress node forwards an output packet toward 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 packet 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. This algorithm is slightly simpler than the median computations described previously in that the egress node need not receive all three copies of a packet prior to forwarding it onward; it need only receive the first two.

VII. Performance Evaluation

In this section we evaluate the performance of *StopWatch* in a variety of tests that highlight the main bottlenecks in our present implementation. Admittedly, our present implementation is largely unoptimized and encounters some stability issues, but our prototype is mature enough to run tests reasonably reliably. We describe some additional detail regarding our implementation that impacts performance in §VII-A, our experimental setup in §VII-B, and our tests and their results in §VII-C–VII-D.

A. Selected implementation details

Our prototype implementation is a modification of Xen version 4.0.2-rc1-pre, amounting to insertions or changes of roughly 1500 SLOC in the hypervisor. In addition, there were about 2000 SLOC insertions and changes to the QEMU device models distributed with that Xen version.

In addition to these changes, we incorporated the multicast implementation OpenPGM (http://code.google.com/p/openpgm/) into the device models in Dom0. OpenPGM is a high-performance implementation of a reliable multicast protocol, specifically of the Pragmatic General Multicast (PGM) specification [44]. In PGM, reliable transmission is accomplished by receivers detecting loss and requesting retransmission of lost data. OpenPGM is used in *StopWatch* for replicating packets destined to a guest VM to all of that VM's replicas and for communication among the VMMs hosting guest VM replicas.

Two additional items are important to understand the results of our experiments in this section. Recall from §V that each VMM proposes (via an OpenPGM multicast) a virtual delivery time for each I/O 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 to the current virtual time of the guest VM. Δ 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 should be minimized since the real time to which Δ translates imposes a lower bound on the latency of the interrupt delivery. (Note that because Δ is specified in virtual time and virtual time can vary in its relationship to real time, the exact real time to which Δ translates can vary during execution.) In our present implementation, the value of Δ used for disk interrupts translates to a real-time delay in the vicinity of 30ms on the platform used in our experiments (see §VII-B). For network interrupts, the value of Δ used translates to roughly 15ms.

A second detail that is important for understanding performance of our prototype is that when, after a VM exit, the VMM determines that the guest VM's virtual time has surpassed the virtual delivery time of an as-yet-undelivered I/O interrupt, the VMM *pauses* the virtual CPU (vCPU) of the guest VM so as to give the device model an opportunity to inject the interrupt before the next VM entry. After taking the necessary steps to inject the interrupt, the device model then *unpauses* the vCPU. Pausing and unpausing the vCPU is a relatively heavyweight operation in Xen; our measurements suggest that simply pausing and unpausing incurs overhead of 0.5ms on average, though we have observed this cost to be more than twice that large, as well. Since pausing and unpausing can happen frequently — e.g., ten times per second or more in some of our tests — this latency can add up.

B. Experimental setup

Our “cloud” consisted of three machines with the same hardware configuration: 4 Intel Core2 Quad Q9650 3.00GHz CPUs, 8GB memory, and 70GB disk. 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. An emulated ATA QEMU disk and a QEMU Realtek RTL-8139/8139C/8139C+ were provided to the guest as its disk and network card. As discussed in §V-B, our implementation of DMA in *StopWatch* is not complete at present, and so the DMA device was disabled. For the same reason, the Advanced Programmable Interrupt Controller (APIC) was also disabled. In each of our tests, we installed an application (e.g., a web server, NFS server, or other benchmarking program) in the guest VM, as will be described later.

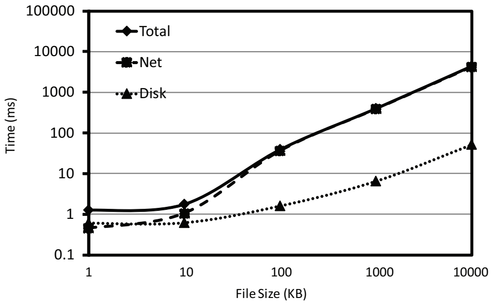
After the guest VM was configured, we used the `xm` save facility provided by Xen to save the running guest state into a file. Instead of restoring this saved state only to one machine, we copied the saved state to our three machines and restored the VM at each. In this way, our three replicas started 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 §V. These three machines were attached to a /24 subnet within our campus network, and as a result, broadcast traffic on the network (e.g., ARP requests) was additionally replicated for delivery as in §V. The volume of 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.

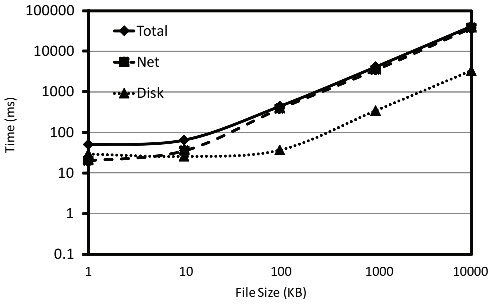
C. Network Services

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 an 802.11 wireless network on our campus.

1) File download: 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 Fig. 4. 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 “Total” curve in Fig. 4(a) shows the average latency for the client to retrieve a file from an unmodified Xen guest VM in the cloud. The “Total” curve in Fig. 4(b) shows the average cost of file retrieval from our *StopWatch* implementation. Every average is for ten runs. Note that both axes are log-scale.



(a) Baseline, without *StopWatch*



(b) *StopWatch*

Figure 4. Average HTTP file-retrieval latency as a function of file size.

To better understand the components of the costs in both the baseline and *StopWatch* cases, we crafted a small program that performs the same function as a web server but that does so in a way that cleanly separates the costs of retrieving the file from disk and of sending the file to the client. More specifically, this program first reads the entire file into a buffer and only then does it send the file to the client in its entirety. By serializing these steps and measuring each individually, we gain a better appreciation for the component costs and *StopWatch*’s impacts on them. The curves marked “Net” in Fig. 4 illustrate the average measured network costs, and the curves marked “Disk” illustrate the disk costs.

Fig. 4 illustrates that for file download, a service running on our current *StopWatch* prototype loses approximately one order of magnitude in download speed. That is, a server running on our *StopWatch* prototype downloaded roughly 10% of the data that a regular web server did in the same amount of time. While the disk access costs increased in *StopWatch* in our experiments in comparison to the baseline, the bottleneck by an order of magnitude or more was the network transmission delay in both the baseline and for

StopWatch. The network performance degradation of *StopWatch* in comparison to the baseline was dominated by the time for delivery of *inbound* packets to the web-server guest VM, i.e., the TCP SYN and ACK messages in the three-way handshake, and then additional acknowledgements sent by the client. Enforcing a median timing on output packets (§VI) adds modest overhead in comparison.

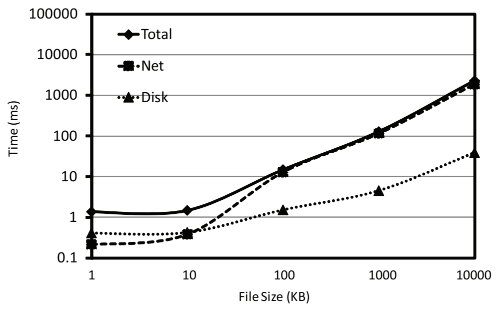
This combination of insights, namely the detriment of inbound packets (mostly acknowledgements) to *StopWatch* file download performance and the fact that these costs so outweigh disk access costs, raises the possibility of recovering file download performance using a transport protocol that utilizes *negative* acknowledgements, so as to minimize unnecessary packets inbound to the web server. Alternatively, a transport protocol with no acknowledgements, such as UDP, could be used; in this case, transmission reliability could instead be enforced at a layer above UDP, and again, a technique using negative acknowledgements would be best. Though TCP does not define negative acknowledgements, transport protocols that implement reliability using negative acknowledgements are widely available, particularly for reliable *multicast* where positive acknowledgements can lead to "ack implosion." Indeed, recall that the PGM protocol specification [44], and so the OpenPGM implementation that we use, ensures reliability using negative acknowledgements.

To illustrate this point, in Fig. 5 we repeat the same experiments as in Fig. 4 but using the Linux utility `udpcast` to transfer the file.[3] Fig. 5(a) shows the performance over unmodified Xen; Fig. 5(b) shows the performance over *StopWatch*. Not surprisingly, Fig. 5(a) shows performance comparable to (but slightly more efficient than) the baseline TCP averages in Fig. 4(a), but rather than losing an order of magnitude, *StopWatch* is *competitive* in Fig. 5(b) with these baseline numbers for files of 100KB or more.

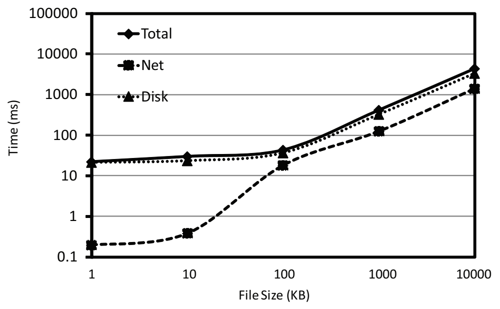
Whereas the network remained the bottleneck in the tests shown in Fig. 5(a), the *disk* was at least as much of a bottleneck in the tests in Fig. 5(b). By eliminating the positive acknowledgements in TCP, the extra networking I/O costs associated with using *StopWatch* were reduced essentially to the median selection by the egress node (see §VI), which were minimal. This left disk I/O as the main bottleneck; since the disk I/O costs were dominated by the networking I/O costs by an order of magnitude in the TCP case (Fig. 4(b)), "falling back" to the disk I/O bottleneck permitted *StopWatch* UDP file transfer (Fig. 5(b)) to perform comparably to the baseline TCP performance in Fig. 4(a).

We reiterate that the performance offered in Fig. 5(b) is not specific to UDP. Rather, this performance should also be achievable with a reliable transport protocol designed to minimize client-to-server messages during file download, as is typical of negative acknowledgment schemes.

[3]We stress that we are not advocating UDP for file retrieval generally but rather are simply demonstrating the advantages for *StopWatch* of file downloading with a protocol that minimizes client-to-server packets. We did not use OpenPGM in these tests since the web site (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 acknowledgements, however.



(a) Baseline, without *StopWatch*



(b) *StopWatch*

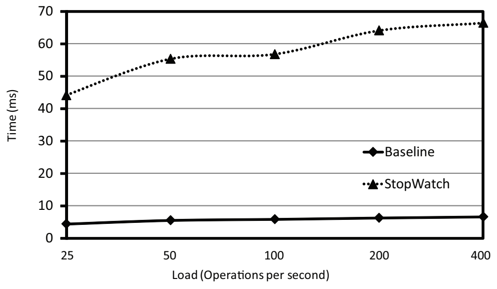
Figure 5. Average `udpcast` file-retrieval latency as a function of file size.

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 and remotely mounted the filesystem exported by the NFS server. We 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 is shown in Fig. 6(a). We obtained the mix file by using `nfsstat` on the NFS server to print its server-side statistics. 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.

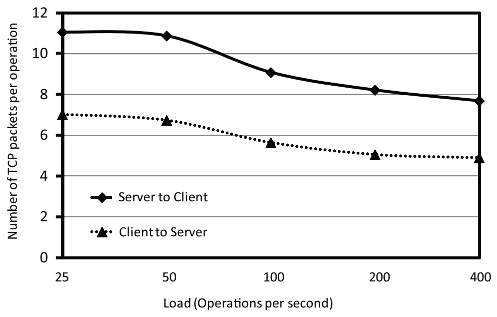
The average latency per operation is shown in Fig. 6(b). In this figure, the horizontal axis is the rate at which operations were submitted to the server; note that this axis grows at an exponential rate. Fig. 6(b) suggests that an NFS server over *StopWatch* incurs roughly a $10\times$ increase in latency 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 Fig. 4.

Operation	%	Operation	%
null	0	getattr	7.93
setattr	11.37	root	0
lookup	24.07	read	32.34
write	11.92	create	12.37
remove	0	link	0
mkdir	0	rmdir	0
readdir	0	fsstat	0

(a) NFS operation distribution



(b) Average latency per operation



(c) Average number of TCP packets per operation

Figure 6. Tests of NFS server using `nhfsstone`

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 is in part due to the fact that *StopWatch* can schedule 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, as shown in Fig. 6(c).

D. Emerging 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 [45]. PARSEC is a diverse set of benchmarks that does not focus on a single application domain, but rather 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.

More specifically, we utilized the following five applications from the PARSEC suite (version 2.1), providing each the "native" input designated for it.

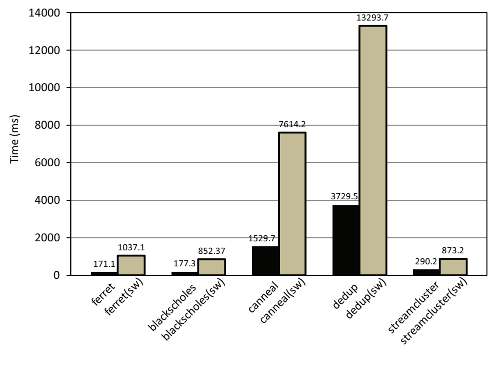
1) `ferret`: This application is representative of next-generation search engines for non-text document data types. In our tests, we configured the application for image similarity search.
2) `blackscholes`: This application calculates option pricing with Black-Scholes partial differential equations and is representative of financial analysis applications.
3) `canneal`: This application is representative of engineering applications and uses simulated annealing to optimize routing cost of a chip design.
4) `dedup`: This application represents next-generation backup storage systems characterized by a combination of global and local compression.
5) `streamcluster`: This application is representative of data mining algorithms for online clustering problems.

Each of these applications involves a variety of activities, including initial configuration, creating a local directory for results, unpacking input files, performing the described computation, and finally cleaning up temporary files.

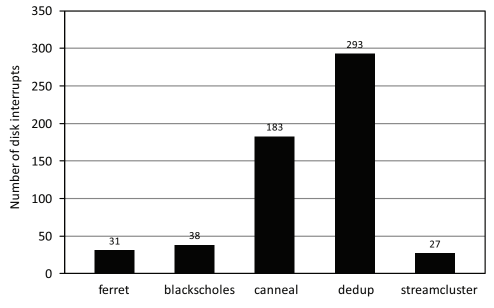
We ran each benchmark ten times within a single guest VM over unmodified Xen, and then ten more times with three guest VM replicas over *StopWatch*. Fig. 7(a) shows the average runtimes of these benchmark applications in the two cases. In this figure, each application is described by a pair of bars; the black bar on the left shows the performance of the application over unmodified Xen, and the beige bar on the right shows the performance of the application over *StopWatch* (and is labeled with a "(sw)" designation). *StopWatch* imposed an overhead between roughly 3× (for `streamcluster`) to 6× (for `ferret`) 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 overwhelmingly due to the overhead involved in intervening on disk I/O (see §V). As shown in Fig. 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.

VIII. Replica Placement in the Cloud

The mechanisms embodied in *StopWatch* are designed to enable a cloud operator to provide defenses against timing side-channel attacks by ensuring that the three replicas of each guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. This imposes constraints on how a cloud operator places guest VM replicas on its machines. In this section we seek to clarify how significant these



(a) Runtimes w/o and w/ *StopWatch*



(b) Disk interrupts per application

Figure 7. Tests of PARSEC applications

placement constraints are, in terms of the provider's ability to best utilize its infrastructure investment. 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 might as well forgo *StopWatch* and simply run each guest VM (non-replicated) in isolation on a separate node. Fortunately, we will see in this section 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, each vertex corresponding to one of the cloud's machines. For every guest VM submitted to the cloud, we characterize the placement of its three replicas as a *triangle* in K_n consisting of the vertices corresponding to the machines on which the replicas are placed and the edges between those vertices. The constraint that the three replicas of each guest VM be coresident with nonoverlapping sets of (replicas of) other VMs can be expressed by requiring that the triangles representing their placements be pairwise *edge-disjoint*. As such, the number k of guest VMs that can simultaneously be run on a cloud of n machines under the constraints of *StopWatch* is the same as the number of edge-disjoint triangles that can be *packed* into K_n. A corollary of a general result due to Horsley [46, Theorem 1.1] is:

Theorem 1. *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 $3k \leq \binom{n}{2}$ and $\binom{n}{2} - 3k \notin \{1, 2\}$; and (ii) if n is even, then k is the largest integer such that $3k \leq \binom{n}{2} - \frac{n}{2}$.*

So, a cloud of n machines using *StopWatch* can simultaneously execute $k = \Theta(n^2)$ guest VMs.

Algorithms for packing edge-disjoint triangles in a graph have previously been studied due to their uses in computational biology (e.g., [47]), yielding practical algorithms for placing triangles to approximate the optimal value of k triangles on K_n to within a constant factor. For example, the greedy approach of arbitrarily placing the next triangle so as to not share any edges with those already placed will successfully place at least $\frac{1}{3}k$ triangles, and more sophisticated augmentation algorithms can achieve at least $\frac{3}{5}k$ in polynomial time [48]. These results immediately translate to algorithms by which a cloud operator using *StopWatch* can place guest VM replicas efficiently.

Of course, these algorithms for packing triangles in graphs were not devised with attention to the nuances of scheduling guest VMs in a cloud. For example, different guest VMs come with different resource demands that must be taken into account in their placement. An interesting direction of future work is to adapt these algorithms to accommodate guest VMs' resource requests as well as the constraints imposed by *StopWatch*.

IX. Future Work

Aside from the direction of future work mentioned at the end of the preceding section, there are some limitations of our existing *StopWatch* prototype that warrant further development. The first is the fact that it supports only uniprocessor guest VMs. As discussed in §II, previous research on replay of multiprocessor VMs (e.g., [33]) should provide a basis for extending our current *StopWatch* prototype beyond uniprocessor VMs, and we are currently investigating this direction. A second direction that we believe presents opportunities for improvement is the performance of our prototype, which is relatively unoptimized at this point. For this reason, we do not believe that the performance evaluation in §VII represents the best performance achievable for *StopWatch*, though it does shed light on the factors that most influence the performance of applications running over *StopWatch*.

We have implicitly assumed in our *StopWatch* implementation — and in many of our descriptions in this paper — that the replicas of each VM are placed on a set of homogeneous machines. Expanding our approach and implementation to heterogeneous machines poses additional challenges that we hope to address in future work. This possibility would also impact the placement algorithms summarized in §VIII, perhaps in a way similar to how diverse workloads would.

A more foundational topic for future research is quantitatively evaluating the degree to which enforcing the timing of events to conform to the median of several replicas, as we do in *StopWatch*, interferes with timing channels. The property is compelling at an intuitive level, we believe, and has precedent in other, related domains, e.g., Byzantine clock synchronization, as discussed in §II, and inference control in statistical databases (e.g., [4]–[6], [49]). We are currently investigating formal bases and empirical frameworks for backing this intuitive appeal. One possible basis for doing so is differential privacy [3], as discussed in §I.

X. Conclusion

We have proposed a novel method of 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 imposing on all replicas the median timing of each observable event among the replicas, we achieve a defense against timing channels that is conceptually akin to differential privacy. 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 overhead induced by our current prototype is large for some activities, we used our evaluation to identify the sources of costs and alternative application designs (e.g., reliable transmission using negative acknowledgements, to support serving files) that can enhance performance considerably. Finally, we identified algorithmic research in graph theory and computational biology that provides a basis for cloud operators to schedule guest VMs under the constraints of *StopWatch* while still utilizing their infrastructure effectively. We envision a mature version of *StopWatch* being a possible basis for the construction of a high-security cloud offering, as would be suitable for supporting communities with significant assurance needs (e.g., military, intelligence, or financial communities).

References

[1] W.-M. Hu, "Reducing timing channels with fuzzy time," in *1991 IEEE Symposium on Security and Privacy*, 1991, pp. 8–20.

[2] N. R. Adam and J. C. Worthmann, "Security-control methods for statistical databases: A comparative study," *ACM Computing Surveys*, vol. 21, no. 4, Dec. 1989.

[3] C. Dwork, "A firm foundation for private data analysis," *Communications of the ACM*, vol. 54, no. 1, Jan. 2011.

[4] J. Domingo-Ferrer and V. Torra, "Median-based aggregation operators for prototype construction in ordinal scales," *International Journal of Intelligent Systems*, vol. 18, no. 6, pp. 633–655, 2003.

[5] V. Torra, "Microaggregation for categorical variables: A median based approach," in *Privacy in Statistical Databases, CASC Project Final Conference*, Jun. 2004, pp. 162–174.

[6] M. E. Kabir and H. Wang, "Microdata protection method through microaggregation: A median-based approach," *Information Security Journal: A Global Perspective*, vol. 20, pp. 1–8, 2011.

[7] Z. S. Xu and Q. L. Da, "An overview of operators for aggregating information," *International Journal of Intelligent Systems*, vol. 18, no. 9, pp. 953–969, 2003.

[8] J. Agat, "Transforming out timing leaks," in *27th ACM Symposium on Principles of Programming Languages*, 2000, pp. 40–53.

[9] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *16th IEEE Computer Security Foundations Workshop*, Jun. 2003, pp. 29–43.

[10] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *22nd IEEE Computer Security Foundations Symposium*, Jul. 2009, pp. 324–335.

[11] B. Köpf and G. Smith, "Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks," in *23rd IEEE Computer Security Foundations Symposium*, Jul. 2010, pp. 44–56.

[12] A. Askarov, A. C. Myers, and D. Zhang, "Predictive black-box mitigation of timing channels," in *17th ACM Conference on Computer and Communications Security*, Oct. 2010, pp. 520–538.

[13] M. H. Kang and I. S. Moskowitz, "A pump for rapid, reliable, secure communication," in *ACM Conference on Computer and Communications Security*, Nov. 1993, pp. 119–129.

[14] J. Giles and B. Hajek, "An information-theoretic and game-theoretic study of timing channels," *IEEE Transactions on Information Theory*, vol. 48, no. 9, Sep. 2002.

[15] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire," in *20th USENIX Security Symposium*, Aug. 2011.

[16] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *18th ACM Conference on Computer and Communications Security*, Oct. 2011.

[17] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, no. 2, pp. 95–114, May 1978.

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul.

1982.

[20] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 1–24, Feb. 1989.

[21] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verissimo, L. Rodrigues, and N. A. Speirs, "The Delta-4 extra performance architecture (XPA)," in *20th International Symposium on Fault-Tolerant Computing*, Jun. 1990, pp. 481–488.

[22] T. C. Bressoud, "TFT: A software system for application-transparent fault tolerance," in *28th International Symposium on Fault-Tolerant Computing*, Jun. 1998, pp. 128–137.

[23] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Enforcing determinism for the consistent replication of multithreaded CORBA applications," in *IEEE Symposium on Reliable Distributed Systems*, Oct. 1999, pp. 263–273.

[24] J. Napper, L. Alvisi, and H. Vin, "A fault-tolerant Java virtual machine," in *2003 International Conference on Dependable Systems and Networks*, Jun. 2003, pp. 425–434.

[25] C. Basile, Z. Kalbarczyk, and R. K. Iyer, "Active replication of multithreaded applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 5, pp. 448–465, May 2006.

[26] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, Feb. 1996.

[27] F. B. Schneider, "Undersanding protocols for Byzantine clock synchronization," Department of Computer Science, Cornell University, Tech. Rep. 87-859, Aug. 1987.

[28] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance for intrusion detection," in *Recent Advances in Intrusion Detection: 8th International Symposium*, 2005, pp. 63–81.

[29] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *15th USENIX Security Symposium*, Aug. 2006.

[30] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *38th IEEE/IFPF International Conference on Dependable Systems and Networks*, Jun. 2008.

[31] D. Gao, M. K. Reiter, and D. Song, "Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 96–110, 2009.

[32] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman, "ReTrace: Collecting execution trace with virtual machine deterministic replay," in *3rd Workshop on Modeling, Benchmarking and Simulation*, Jun. 2007.

[33] G. W. Dunlap, D. G. Lucchetti, P. M. Chen, and M. A. Fetterman, "Execution replay of multiprocessor virtual machines," in *4th ACM Conference on Virtual Execution Environments*, Mar. 2008, pp. 121–130.

[34] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++," in *24th ACM Conference on Object Oriented Programming, Systems, Languages and Applications*, Oct. 2009, pp. 81–96.

[35] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," in *15th Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010, pp. 53–64.

[36] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

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

[38] J. C. Wray, "An analysis of covert timing channels," in *1991 IEEE Symposium on Security and Privacy*, 1991, pp. 2–7.

[39] G. Popek and C. Kline, "Verifiable secure operating system software," in *AFIPS National Computer Conference*, 1974, pp. 145–151.

[40] *Timekeeping in VMware Virtual Machines*, VMWare Inc., May 2010.

[41] 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, "Intel virtualization technology," *IEEE Computer*, vol. 38, no. 3, pp. 48–56, May 2005.

[42] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Oct. 2011.

[43] D. Dolev and H. R. Strong, "Authenticated algorithms for Byzantine agreement," *SIAM Journal of Computing*, vol. 12, pp. 656–666, 1983.

[44] T. Speakman, et al., "PGM reliable transport protocol specification," Request for Comments 3208, Internet Engineering Task Force, Dec. 2001.

[45] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[46] D. Horsley, "Maximum packing of the complete graph with uniform length cycles," *Journal of Graph Theory*, vol. 68, no. 1, pp. 1–7, Sep. 2011.

[47] V. Bafna and P. A. Pevzner, "Genome rearrangements

and sorting by reversals," *SIAM Journal on Computing*, vol. 25, no. 2, pp. 272–289, Apr. 1996.

[48] T. Feder and C. Subi, "Packing edge-disjoint triangles in given graphs," last retrieved from http://theory.stanford.edu/~tomas/triclique.ps on 14 Nov. 2011.

[49] A. Roth and T. Roughgarden, "Interactive privacy via the median mechanism," in *42nd ACM Symposium on Theory of Computing*, Jun. 2010, pp. 765–774.