Mitigating Timing Channels in Clouds using StopWatch[*]

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 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 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 could have been observed in the absence of any other individual, coresident VM. 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.

1 Introduction

Implicit timing-based information flows potentially threaten the use of clouds for very sensitive computations. In an "infrastructure as a service" (IaaS) cloud, such an attack would 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 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., [25, 45]). 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., [1]). 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" [18] 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. 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, its timings 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 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.[1]

[*]This technical report subsumes TR11-010 [34].

[1]The median can be viewed as "microaggregating" the timings to

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). 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 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 extensively evaluate the performance of our *StopWatch* prototype for supporting web service (file downloads), network file systems, and various types of computations. Our analysis shows that performance costs of *StopWatch* can range up to $3\times$ for network-intensive applications, and it further 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 that minimizes client-to-server acknowledgements (or unreliable transport with no acknowledgements, as in UDP) can dramatically improve file download latencies versus TCP 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 their amounts of disk I/O.

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: First, 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 timings of 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. Third, 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.

confound inferences from them (c.f., [16, 43, 27]). This analogy suggests the possibility of using other microaggregation functions, as well, of which there are many [49].

2 Related Work

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., [2, 50]), 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., [30, 31]), but we seek an approach that applies to general computations.

Askarov et al. [3] 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., [25, 2, 50, 45]) and external (e.g., [28, 22, 3, 23, 51]) 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. This offers a more principled defense than randomly perturbing the timings of events observable at or from an nonreplicated attacker VM (e.g., [25]). Random noise does not asymptotically eliminate timing channels [3], in part because a distribution from which to draw the randomness must be chosen without reference to an execution in the absence of the victim — i.e., how the execution "should have" looked. *StopWatch* uses replication and careful replica placement (in terms of the other VMs with which each replica coresides) exactly to provide such a reference.

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 [32, 41] to mask Byzantine faults [33] 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 §3 and §6). 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., [11, 6, 12, 36, 35, 7]), but not all (e.g., [13]), 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 [40, §5.2]).

Aside from replication for fault tolerance, replication has been explored to detect server penetration [20, 14, 37, 21]. 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., [48, 17]) 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., [17]) 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., [9, 8]) or the OS (e.g., [4]) are less relevant to our goals, as they are not easily utilized by an IaaS cloud provider that accepts arbitrary VMs for execution.

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., [39]), 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 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 [47], 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 a "clock," a timing attack simply involves measuring one clock using another. Wray identified four possible clock sources in conventional computers [47]:

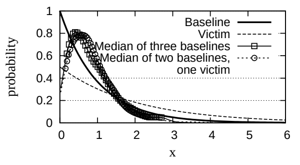
- TL: the "CPU instruction-cycle clock", i.e., a clock constructed by executing a simple timing loop;
- Mem: the memory subsystem (e.g., data/instruction fetches);
- IO: the I/O subsystem (e.g., network, disk, and DMA interrupts); and
- RT: real-time clocks provided by the hardware platform (e.g., time-of-day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

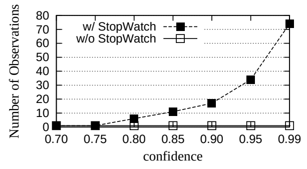
More specifically, to interfere with IO clocks, *StopWatch* replicates each attacker VM (i.e., every VM, since we do not presume to know which ones are attacker VMs) threefold so that the three replicas of a guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. Then, when determining the timing with which an event is made available to each replica, the median timing value of the three is adopted. We justify the median below, and as discussed in §2, we view *StopWatch*'s use of medians in addressing IO clocks as one of our primary innovations. *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 [38]. Optionally, *StopWatch* adjusts virtual time periodically using the median real time of the three replicas, thereby roughly synchronizing their views of real time with actual real time to a degree.

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 the arrival of packets that the attacker VM sends. 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.

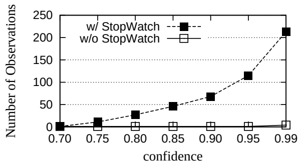
Permitting only the median timing of an event to be observed limits the information that an attacker VM can glean from being co-located with a victim VM of interest, because the distribution of the median timings substantially dampens the visibility of a victim's activities. To see how, consider a victim VM that induces timings that are exponentially distributed with rate λ', versus a baseline (i.e., non-victim) exponential distribution with rate $\lambda > \lambda'$.[2] Fig. 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'; e.g., $\lambda = 1$ and $\lambda' = 1/2$ in the example in Fig. 1(a). In this case, to 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 (Fig. 1(b)). This improvement becomes even more pronounced if λ and λ' are closer; the cases $\lambda = 1$, $\lambda' = 2/3$ and $\lambda = 1$, $\lambda' = 10/11$ are shown in Figs. 1(c) and 1(d), respectively.



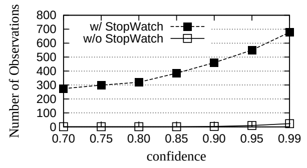
(a) Distribution of median; $\lambda' = 1/2$



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



(c) Observations needed to detect victim; $\lambda' = 2/3$



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

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

Of course, in terms of absolute numbers of observa-

[2] It is not uncommon to model packet inter-arrival time, for example, using an exponential distribution (e.g., [29]).

tions needed to detect the victim VM with confidence, this assessment may be very conservative, since the attacker would face numerous pragmatic difficulties that we have not modeled here (e.g., migration of VMs between cores). Moreover, detecting the victim VM is only the first step of extracting useful information (e.g., a cryptographic key) from it. But even this simple example shows the power of disclosing only median timings of three VM replicas, and in §5.2 we will repeat this illustration using actual message traces.

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 [38]. That is, after the VM executes *instr* instructions, the virtual time observed from within the VM is

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

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.

Optionally, the VMMs can adjust *start* and *slope* periodically, e.g., after the replicas execute an "epoch" of I instructions, to coarsely synchronize *virt* and real time. For example, after the k-th 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

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

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

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

4.2 Implementation in Xen

Real-time clocks on 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 a count of processor ticks since reset.

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 [46]. 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 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 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 [44]. 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 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 [26, §29.3.2]. For VM exits caused by guest VM execution, the VMM injects any needed timer interrupts on the next VM entry.

`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 (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.

Reading counters. The guest can also observe real time from 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. 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 (1), 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 §5.1, and then in §5.2 we describe our implementation of this strategy in *StopWatch*.

5.1 Strategy

The method described in §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 §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.

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 a constant Δ_d to determine a "delivery time" for the interrupt, i.e., at virtual time $V + \Delta_d$, and initiates the corresponding I/O activities (disk access or DMA transfer). The constant 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 §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 [25, §4.2.2]).

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 §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 §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. 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. In distributed computing parlance, we thus assume a *synchronous* system, i.e., there are known bounds on processor execution rates and message delivery times. The synchronous model has been widely used to develop and deploy distributed protocols (e.g., [15]).

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 §4.2) that occurs at a virtual time at least as large as that median value.[4] 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., see [25, §4.2.2]).

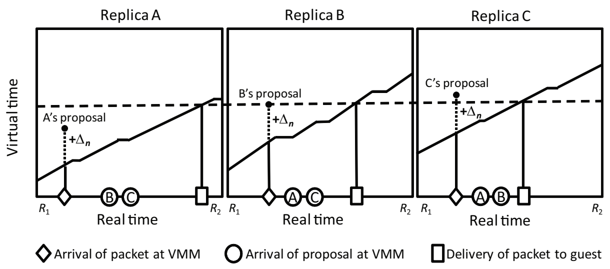


Figure 2: Delivering a packet to guest VM replicas.

The process of determining the delivery time of a network packet to guest VMs replicas is pictured in Fig. 2.

[4]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, e.g., copying the state of another replica.

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. (See Fig. 3.)

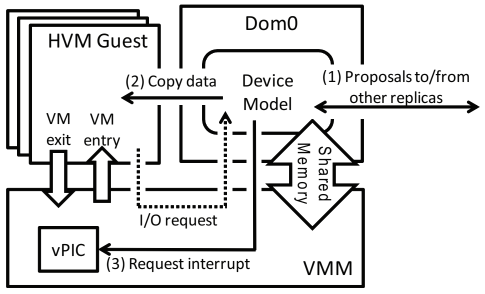
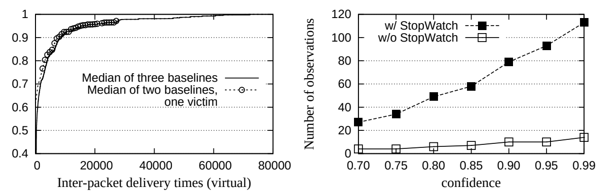


Figure 3: Emulation of I/O devices in *StopWatch*. "I/O request" present only for disk I/O.

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 [26, §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_n 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 3). A memory region shared between Dom0 and the VMM allows device models in Dom0 to read guest virtual time, which is computed and



(a) Distribution of median (CDF)   (b) Observations needed to detect victim

Figure 4: Virtual inter-packet delivery times with coresident victim (“two baselines, one victim”) and in a run where no replica was coresident with a victim (“three baselines”)

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

In Fig. 4(a) we show the distribution of virtual inter-packet delivery times in an actual run with an active victim, in comparison to the virtual delivery times with no victim present. This plot is directly analogous to that in Fig. 1(a) but is generated from a real trace and shows the distribution as a CDF for ease of readability. Fig. 4(b) shows the number of observations needed to distinguish the victim and no-victim distributions in Fig. 4(a) as a function of the desired confidence. This figure is analogous to that in Fig. 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.

Disk and DMA emulation. The emulation of the IDE disk and DMA devices is similar to the network card emulation above. *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 External Observers

The mechanisms described in §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 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 separate “egress node” that is dedicated for this purpose, analogous to the “ingress node” that replicates every network packet destined to the guest VM to the VM’s replicas (see §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 over TCP. 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. This algorithm is slightly simpler than the median calculations described previously, since the egress node need not receive all three copies of a packet prior to forwarding it; it need only receive the first two.

7 Performance Evaluation

In this section we evaluate the performance of our *StopWatch* prototype. We present additional implementation details that impact performance in §7.1, our experimental setup in §7.2, and our tests and their results in §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 1500 source lines of code (SLOC) in the hypervisor. There were also about 2000 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 [42]. 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.

Recall from §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 differentials in the arrivals of packets destined to the guest VM at its three replicas' VMMs and the delays for delivering each VMM's proposed virtual delivery time to the others. For the platform used in our experiments (see §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 12-15ms sufficed to meet the above criteria. The analogous offset Δ_d for determining the virtual delivery time for disk and DMA interrupts translates to roughly 15-20ms.

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

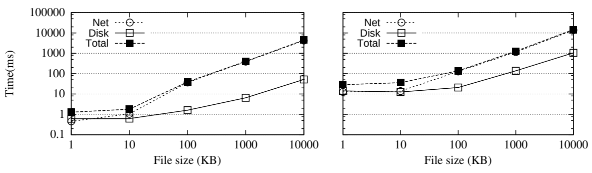
After the guest VM was configured, we copied it to our three machines and restored the VM at each. In this way, our three replicas started running from the same state. In addition, we copied the disk file to all three machines to provide identical disk state to the three replicas.

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 §5. 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 replicated for delivery as in §5. 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.

7.3 Network Services

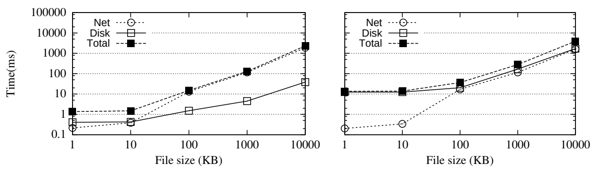
In this section we describe tests involving network services deployed on the cloud. In all of our tests, our client that interacted with the cloud-resident service was a Lenovo T400 laptop with a dual-core 2.8GHz CPU and 2GB memory attached to an 802.11 wireless network on our campus.

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. 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 "Total" curve in Fig. 5(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. 5(b) shows the average cost of file retrieval from our *StopWatch* implementation. Every average is for ten runs. Note that both axes



(a) Without *StopWatch* (b) With *StopWatch*

Figure 5: Average HTTP file-retrieval latency.



(a) Without *StopWatch* (b) With *StopWatch*

Figure 6: Average `udpcast` file-retrieval latency.

are log-scale.

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 "Net" curves in Fig. 5 show the average measured network costs, and the "Disk" curves show the disk costs.

Fig. 5 shows that for file download, a service running on our current *StopWatch* prototype loses roughly 3$\times$ in download speed for files of 100KB or larger. 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 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, 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 (§6) 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 minimizes packets inbound to the web server, e.g., using negative acknowledgements or forward error correction. Alternatively, an unreliable transport protocol with no acknowledgements, such as UDP, could be used; transmission reliability could then be enforced at a layer above UDP using negative acknowledgements or forward error correction. Though TCP does not define negative acknowledgements, transport protocols that implement reliability using them are widely available, particularly for *multicast* where positive acknowledgements can lead to "ack implosion." Indeed, recall that the PGM protocol specification [42], and so the OpenPGM implementation that we use, ensures reliability using negative acknowledgements.

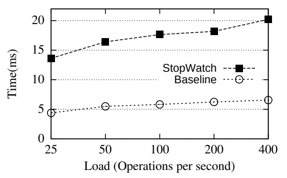
To illustrate this point, in Fig. 6 we repeat the experiments in Fig. 5 but using the Linux utility `udpcast` to transfer the file.[5] Fig. 6(a) shows the performance using unmodified Xen; Fig. 6(b) shows the performance using *StopWatch*. Not surprisingly, Fig. 6(a) shows performance comparable to (but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

Whereas the network remained the bottleneck in the tests shown in Fig. 6(a), the *disk* was at least as much of a bottleneck in the tests in Fig. 6(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 §6), which were minimal. Disk I/O remained as the main bottleneck, but it imposed an order of magnitude less overhead than the networking I/O costs previously had (Fig. 5(b)). This reduction, in turn, permitted *StopWatch* UDP file transfer (Fig. 6(b)) to perform comparably to the baseline TCP performance in Fig. 5(a).

We reiterate that the performance offered in Fig. 6(b) is not specific to UDP. 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 and protocols using forward error correction.

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

[5]We are not advocating UDP for file retrieval generally but rather are simply demonstrating the advantages for *StopWatch* of 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) Average latency per op

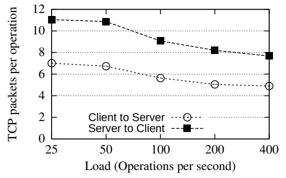
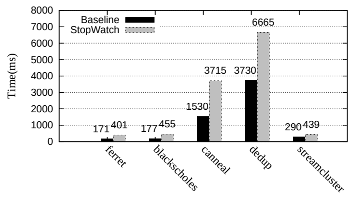

(b) Average packets per op

Figure 7: Tests of NFS server using `nhfsstone`


(a) Average runtimes

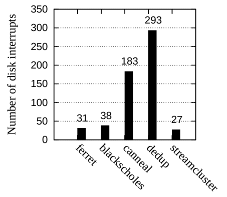

(b) Disk interrupts

Figure 8: Tests of PARSEC applications

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.[6] 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. 7(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. Fig. 7(a) suggests that an NFS server over *StopWatch* incurs roughly a 3$\times$ 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. 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 (Fig. 7(b)).

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 [10]. 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 non-text 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 within a single guest VM over unmodified Xen, and then ten more times with three guest VM replicas over *StopWatch*. Fig. 8(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*. *StopWatch* imposed an overhead of at most 2.57$\times$ (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 overwhelmingly due to the overhead involved in intervening on disk I/O (see §5). As shown in Fig. 8(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.

8 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 imposes constraints on how a

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

cloud operator places guest VM replicas on its machines. In this section we seek to clarify how significant these placement constraints are, 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 (non-replicated) in isolation on a separate node. Fortunately, we will see 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 k 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 [24, 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., [5]), yielding practical algorithms for placing triangles to approximate the optimal value of k triangles on K_n to within a constant factor. For example, a greedy approach 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 [19]. 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 do not account for the nuances of scheduling guest VMs in a cloud. For example, different guest VMs come with different resource demands. A direction for future work is to adapt these algorithms to accommodate guest VMs' resource needs as well as the constraints imposed by *StopWatch*.

9 Collaborating Attacker VMs

Our discussion so far has not explicitly addressed the possibility of attacker VMs collaborating to mount 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 the reasoning in Fig. 1 and its confirmation in Fig. 4 suggest 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 by *StopWatch*. 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.

10 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 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 $3\times$ for networking applications, 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 results in graph theory and computational biology that provides a basis for clouds 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 facility, as would be suitable for supporting communities with significant assurance needs (e.g., military, intelligence, or financial communities).

An important topic for future work is extending *StopWatch* to support multiprocessor guest VMs. As discussed in §2, previous research on replay of multiprocessor VMs (e.g., [17]) should provide a basis for extending our current *StopWatch* prototype, and we are currently investigating this direction. A second direction for improvement is that 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 §8, perhaps in a way similar to how diverse workloads would.

11 References

[1] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: A comparative study. *ACM Computing Surveys*, 21(4), Dec. 1989.

[2] J. Agat. Transforming out timing leaks. In *27th ACM Symposium on Principles of Programming Languages*, pages 40–53, 2000.

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

[4] 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.

[5] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, Apr. 1996.

[6] 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*, pages 481–488, June 1990.

[7] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):448–465, May 2006.

[8] 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*, pages 53–64, Mar. 2010.

[9] 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*, pages 81–96, Oct. 2009.

[10] C. Bienia. *Benchmarking modern multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[11] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.

[12] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *28th International Symposium on Fault-Tolerant Computing*, pages 128–137, June 1998.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, Feb. 1996.

[14] 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.

[15] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3), Aug. 1996.

[16] J. Domingo-Ferrer and V. Torra. Median-based aggregation operators for prototype construction in ordinal scales. *International Journal of Intelligent Systems*, 18(6):633–655, 2003.

[17] 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*, pages 121–130, Mar. 2008.

[18] C. Dwork. A firm foundation for private data analysis. *Communications of the ACM*, 54(1), Jan. 2011.

[19] 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.

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

[21] 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*, 6(2):96–110, 2009.

[22] J. Giles and B. Hajek. An information-theoretic and game-theoretic study of timing channels. *IEEE Transactions on Information Theory*, 48(9), Sept. 2002.

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

[24] D. Horsley. Maximum packing of the complete graph with uniform length cycles. *Journal of Graph Theory*, 68(1):1–7, Sept. 2011.

[25] W.-M. Hu. Reducing timing channels with fuzzy time. In *1991 IEEE Symposium on Security and Privacy*, pages 8–20, 1991.

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

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

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

[29] T. Karagiannis, M. Molle, M. Faloutsos, and A. Broido. A nonstationary Poisson view of Internet traffic. In *INFOCOM 2004*, pages 1558–1569, Mar. 2004.

[30] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *22nd IEEE Computer Security Foundations Symposium*, pages 324–335, July 2009.

[31] B. Köpf and G. Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *23rd IEEE Computer Security Foundations Symposium*, pages 44–56, July 2010.

[32] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2(2):95–114, May 1978.

[33] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[34] P. Li, D. Gao, and M. K. Reiter. StopWatch: Toward "differentially private" timing for cloud executions. Technical Report TR11-010, Department of Computer Science, University of North Carolina at Chapel Hill, 2011.

[35] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant Java virtual machine. In *2003 International Conference on Dependable Systems and Networks*, pages 425–434, June 2003.

[36] 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*, pages 263–273, Oct. 1999.

[37] 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*, June 2008.

[38] G. Popek and C. Kline. Verifiable secure operating system software. In *AFIPS National Computer Conference*, pages 145–151, 1974.

[39] 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*, pages 199–212, 2009.

[40] F. B. Schneider. Undersanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Aug. 1987.

[41] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), Dec. 1990.

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

[43] V. Torra. Microaggregation for categorical variables: A median based approach. In *Privacy in Statistical Databases, CASC Project Final Conference*, pages 162–174, June 2004.

[44] 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*, 38(3):48–56, May 2005.

[45] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen. In *ACM Cloud Computing Security Workshop*, Oct. 2011.

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

[47] J. C. Wray. An analysis of covert timing channels. In *1991 IEEE Symposium on Security and Privacy*,

pages 2–7, 1991.

[48] 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*, June 2007.

[49] Z. S. Xu and Q. L. Da. An overview of operators for aggregating information. *International Journal of Intelligent Systems*, 18(9):953–969, 2003.

[50] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

[51] 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.