REPLICATION AND PLACEMENT FOR SECURITY IN DISTRIBUTED SYSTEMS

Peng Li

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2014

Approved by:

James H. Anderson

Peter M. Chen

Debin Gao

Kevin Jeffay

Michael K. Reiter, Chair

ABSTRACT

Peng Li: Replication and placement for security in distributed systems
(Under the direction of Michael K. Reiter)

In this thesis we show how the security of replicated objects in distributed systems, in terms of either the objects' confidentiality or availability, can be improved through the placement of objects' replicas so as to carefully manage the nodes on which objects' replicas overlap.

In the first part of this thesis we present *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 and threaten confidentiality of victims' data. *StopWatch* triplicates each cloud-resident guest virtual machine (VM) and places replicas so that the three replicas of a guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. *StopWatch* uses the timing of I/O events at a VM's replicas collectively to determine the timings observed by each one or by an external observer, so that observable timing behaviors are similarly likely in the absence of any other individual, coresident VM. We detail the design and implementation of *StopWatch* in Xen, evaluate the factors that influence its performance, demonstrate its advantages relative to alternative defenses against timing side-channels with commodity hardware, and address the problem of placing VM replicas in a cloud under the constraints of *StopWatch* so as to still enable adequate cloud utilization.

We then explore the problem of placing object replicas on nodes in a distributed system to maximize the number of objects that remain available when node failures occur. In our model, failing (the nodes hosting) a given threshold of replicas is sufficient to disable each object, and the adversary selects which nodes to fail to minimize the number of objects that remain available. We specifically explore placement strategies based on combinatorial structures called t-packings; provide a lower bound for the object availability they offer; show that these placements offer availability that is c-competitive with optimal; and propose an efficient algorithm for computing combinations of t-packings that maximize their availability lower bound. We compare the availability offered by our approach to that of random replica placement, owing to the popularity of the latter approach in previ-

ous work. After quantifying the availability offered by random replica placement in our model, we show that our combinatorial strategy yields placements with better availability than random replica placement for many realistic parameter values. Finally, we provide parameter selection strategies to concretely instantiate our schemes for different system sizes.

To my family in China, my wife Qing, and my son Lucas.

ACKNOWLEDGEMENTS

The completion of this dissertation would not have been possible without the guidance of my advisors, Prof. Mike Reiter and Prof. Debin Gao, whom I feel extremely fortunate to have met and worked with and owe all my gratitude to. Throughout the last seven years, weekly meetings with them have shaped the pace of my life and marked the path of my growth both as a researcher and as a person. One afternoon back in 2007, a phone call from Debin brought me to Singapore, and the days I spent there working with him were beyond pleasant and memorable. Through Debin I met Mike, simply the best mind I have had the fortune to work with, not only because of his masterful understanding and insight of the field, prompt and wise decisions he has made on all matters, always-correct direction and advise, but also because of his willingness to share frustrations when things turn out in an unexpected way and his embodiment of something beyond an academic advisor. His high expectation kept me awake solving problems in many stressful nights, but it also lifted me up to stand on a higher ground to watch around. For these many years, "Mike and Debin" have been the most frequent openning of the emails I sent and the most reliable "oracles" for me to count on.

I would also like to thank Dr. James Anderson, Dr. Peter Chen, and Dr. Kevin Jaffey for serving on my dissertation committee. I am very grateful for all of them to take time from their busy schedules to hold meetings with me and providing invaluable feedbacks.

With all my friends in the security lab and in Chapel Hill, I do not feel I am alone. I appreciate gratefully for their help in my academic studies and in my life.

One of the luckiest things that have ever happened to me was meeting with my wife Qing, who moved to the States with all trust in me and supports me no matter what happens. She was born with the abilities of viewing things positively, of being kind to and thinking good of other people, and of discovering amazement in everyday life. I am the happiest man having her aside.

Studying and living abroad has been not easy, luckily I have my parents, my brother, and all my family support me unconditionally. All I did and will do is for them.

PREFACE

Object replication, and careful placement of those replicas so as to manage the overlap between any two objects' replicas, can be used to support security goals in distributed systems. This thesis demonstrates two specific uses of replication and placement for this purpose, namely to limit side-channel information leakage between virtual machines in compute clouds and to improve the availability of objects despite targeted failures of computers that host their replicas.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

Traditionally, the topic of computer security has been characterized as protecting three attributes of data (and/or computation) (e.g., [40, Chapter 1]): *confidentiality* or, in other words, that the data is disclosed only to whom the data owner intends; *integrity*, or that the data is modified only in intended ways and by intended parties; and *availability*, so that the data is accessible when required and with the performance expected. Security is a field that adapts to include new computer and data misuses as they become known, and some of these misuses (e.g., combating spam email) stretch the above characterization of computer security. Still, this characterization is adequate for the discussions of primary interest in this thesis.

It has long been argued that the goals of confidentiality, integrity and availability are themselves in conflict, in the sense that availability focuses on ensuring data's accessibility, whereas confidentiality and integrity seek to limit its accessibility (to unintended disclosure and update, respectively). This tension has specifically been highlighted in the use of data *replication*, since replicating data to potentially far-flung locations might greatly enhance availability but put the data at risk of unintended disclosure or modification (e.g., [83]). Conversely, keeping the data in a high-security vault might enhance its confidentiality and integrity, but it might not be easily accessible when needed or might be destroyed in a fire inside the vault, hurting its availibility.

There is a long history of research focused on striking a balance between availability by replication on the one hand, and confidentiality and integrity of the replicated data/computation on the other. Examples include the division of certificate authorities into online (highly available) and offline (and hence more secure) components (e.g., [56]), data storage that combines redundancy for data availability with cryptographic techniques to enhance integrity and/or confidentiality (e.g., [46]), and the entire field of Byzantine fault-tolerant computation (see, e.g., [55, 73, 24, 18, 45] and citations therein) to balance integrity and availability.

Despite this attention, we show in this thesis that there is another facet of replication that, to our knowledge, has not yet been exploited in the context of security but that provides a new opportunity to explore the design space of secure systems—namely, the manner in which replicas are placed on nodes. More specifically, in this thesis we explore two applications of the general idea of constraining how the replicas of different data/computation objects overlap on nodes to improve security for the objects. Specifically, this thesis leverages replica placement in two ways:

- In Chapter 2, we consider the problem of protecting virtual machines (VMs) submitted to public compute clouds from the inference of their secrets by other VMs utilizing timing-based side-channel attacks (e.g., [97]). We develop a strategy for the cloud to execute VMs that involves replicating each VM and placing its replicas so that they overlap (reside on the same host as) a limited number of replicas of any other VM. Then, by ensuring that the timing of each event observable by any VM is an aggregation of the timings of this event observable by (the machines hosting) its replicas, the limited-overlap policy ensures that any (victim) VM can influence the timing of events observable by another (attacker) VM only minimally. We show that timing-based side-channels available to an attacker VM are thereby substantially mitigated. To our knowledge, this work is noteworthy in demonstrating how replication can *improve* confidentiality of data by illustrating a scenario in which replication supports confidentiality.

- In the context of using replication for availability, the previous work that considered replica placement did so only in scenarios where computers fail probabilistically. Protecting the *security* of a system entails considering intelligent attackers, however, in particular ones whose behaviors may not be characterized by a known distribution or who can target a system adaptively. Therefore, in Chapter 3 we study the problem of maximizing the availability of replicated objects against an attacker who can disable computers in a targeted fashion, with knowledge of where object replicas are placed and limited only by a budget on the total number of computers it can disable. We show how carefully managing overlaps in the placement of object replicas can substantially enhance object availability against this type of targeted attacker.

The above contributions do not eliminate the aforementioned tensions between replication on the one hand and confidentiality and integrity on the other. However, they do provide new insights into unchartered parts of the tradeoff space. Specifically, they provide new ways of using replication and specifically replica placement to enhance a singular security goal—improved confidentiality via timing side-channel defense in the first case above, and improved availability against targeted attacks in the second—that might be possible to leverage in conjunction with other technologies previously described (though we leave this exploration to future work). Together, we believe that these works add a new dimension to previous thinking about security mechanisms for replicated systems, namely that replica placement is a critical factor influencing the utility of replication for both conventional purposes (availability) and unconventional ones (confidentiality) in systems subject to attack.

CHAPTER 2 MITIGATING ACCESS-DRIVEN TIMING CHANNELS IN CLOUDS USING *STOPWATCH*

Implicit timing-based information flows threaten the use of clouds for very sensitive computations. In an "infrastructure as a service" (IaaS) cloud, such an attack could be mounted by an attacker submitting a virtual machine (VM) to the cloud that times the duration between events that it can observe, to make inferences about a *victim* VM with which it is running simultaneously on the same host but otherwise cannot access. Such "access-driven" attacks [97] were first studied in the context of timing-based *covert channels*, in which the victim VM is infected with a Trojan horse that intentionally signals information to the attacker VM by manipulating the timings that the attacker VM observes. Of more significance in modern cloud environments, however, are timing-based *side channels*, which leverage the same principles to attack an uninfected but oblivious victim VM (e.g., [74, 97]).

In this chapter we propose an approach to defend against these timing attacks and a system, called *StopWatch*, that implements this method for IaaS clouds. A timing side-channel can arise whenever an attacker VM uses an event sequence it observes to "time" another, independent event sequence that might reflect the victim VM's behavior [88]. *StopWatch* is thus designed to systematically remove independence of observable event sequences where possible, first by making all real-time clocks accessible from a guest VM to be determined instead by the VM's own execution.

To address event sequences on which it cannot intervene this way, namely for input/output (I/O) events, *StopWatch* alters I/O timings observed by the attacker VM to mimic those of a *replica* attacker VM that is *not* coresident with the victim. Since *StopWatch* cannot identify attackers and victims *a priori*, realizing this intuition in practice requires replicating each VM on multiple hosts and enforcing that the replicas are coresident with nonoverlapping sets of (replicas of) other VMs — so that, in particular, at most one attacker VM replica is coresident with a replica of the victim VM. *StopWatch* then delivers any I/O event to each attacker VM replica at a time determined by

"microaggregating" the delivery times planned by the VMMs hosting those replicas. Specifically, *StopWatch* uses three replicas per VM that coreside with nonoverlapping sets of (replicas of) other VMs and microaggregates the timing of I/O events by taking their median across all three replicas. (Two replicas per VM seems not to be enough: one might be coresident with its victim, and by symmetry, its I/O timings would necessarily influence the timings imposed on the pair.) Even if the median timing of an I/O event is that which occurred at an attacker replica that is coresident with a victim replica, timings both below and above the median occurred at attacker replicas that do not coreside with the victim.

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

We evaluate the performance of our *StopWatch* prototype for supporting web service (file downloads) and various types of computations. Our analysis shows that the latency overhead of *StopWatch* is less than $2.8\times$ even for network-intensive applications. We also identify adaptations to a service that can vastly increase its performance when run over *StopWatch*, e.g., making file download over *StopWatch* competitive with file download over unmodified Xen. For computational benchmarks, the latency induced by *StopWatch* is less than $2.3\times$ and is directly correlated with their amounts of disk I/O. Overall, the latency overhead of *StopWatch* is qualitatively similar to other modern systems that use VM replication for other reasons (e.g., [25]). Moreover, we demonstrate that *StopWatch* can substantially outperform competing defenses against timing side-channel attacks, namely adding uniformly random noise to event timings or running VMs on shared hardware in a time-slicing fashion.

We also study the impact of *StopWatch* on cloud utilization, i.e., how many guest VMs can be simultaneously executed on an infrastructure of n machines, each with a capacity of c guest VMs, under the constraint that the three replicas for each guest VM coreside with nonoverlapping sets

of (replicas of) other VMs. We show that for any $c \leq \frac{n-1}{2}$, $\Theta(cn)$ guest VMs (three replicas of each) can be simultaneously executed; we also identify practical algorithms for placing replicas to achieve this bound. We extend this result to $\Theta(\frac{cn}{d_{max}})$ guest VMs when guest VMs can place different demands, up to d_{max}, on machine resources of capacity c. These results distinguish *StopWatch* from the alternative of simply running each guest VM on a separate computer, which permits simultaneous execution of only n guest VMs.

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

The rest of this chapter is structured as follows. We describe related work in Section 2.1. We provide an overview of the design of *StopWatch* in Section 2.2 and detail how we address classes of internal "clocks" used in timing attacks in Section 2.3 and Section 2.4. In Section 2.5, we then discuss how *StopWatch* extends to address richer attacks involving collaborators external to the cloud or collaborative attacker VMs. We evaluate performance of our *StopWatch* prototype in Section 2.6. We extend this evaluation to provide a comparison to other holistic timing side-channel defenses in Section 2.7. Section 2.8 treats the replica placement problem that would be faced by cloud operators using *StopWatch*, and we conclude in Section 2.9.

2.1 Related Work

2.1.1 Timing Channel Defenses

Defenses against information leakage via timing channels are diverse, taking numerous different angles on the problem. Research on type systems and security-typed languages to eliminate timing attacks offers powerful solutions (e.g., [3, 94, 96]), 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., [82]) or as enabled by specific hardware components (e.g., [72, 54]), but we seek an approach that is comprehensive.

Askarov et al. [4] 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., [49, 3, 94, 85]) and external (e.g., [51, 39, 4, 42, 95]) timing channels have received significant attention individually, though to our knowledge, *StopWatch* is novel in addressing access-driven timing channels through a combination of both techniques. *StopWatch* incorporates internal defenses to interfere with an attacker's use of real-time clocks or "clocks" that it might derive from the I/O subsystem. In doing so, *StopWatch* imposes determinism on uniprocessor VMs and then uses this feature to additionally build an effective external defense against such attacker VMs.

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

better than uniformly random noise allows for the same protection, as the distinctiveness of victim behavior increases.

2.1.2 Replication

To our knowledge, *StopWatch* is novel in utilizing replication for timing channel defense. That said, replication has a long history that includes techniques similar to those we use here. For example, state-machine replication to mask Byzantine faults [78] 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 Section 2.2 and Section 2.5.1). To ensure that correct replicas return the same responses, these systems enforce the delivery of requests to replicas in the same order; moreover, they typically assume that replicas are deterministic and process requests in the order they are received. *Enforcing* replica determinism has also been a focus of research in (both Byzantine and benignly) fault-tolerant systems; most (e.g., [13, 64, 6]), but not all (e.g., [15]), do so at other layers of the software stack than *StopWatch* does.

More fundamentally, to our knowledge all prior systems that enforce timing determinism across replicas permit one replica to dictate timing-related events for the others, which does not suffice for our goals: that replica could be the one coresident with the victim, and so permitting it to dictate timing related events would simply "copy" the information it gleans from the victim to the other replicas, enabling that information to then be leaked out of the cloud. Rather, by forcing the timing of events to conform to the median timing across three VM replicas, at most one of which is coresident with the victim, the enforced timing of each event is either the timing of a replica not coresident with the victim or else between the timing of two replicas that are not coresident with the victim. This strategy is akin to ones used for Byzantine fault-tolerant clock synchronization (e.g., see [77, Section 5.2]) or sensor replication (e.g., see [78, Section 5.1]), though we use it here for information hiding (versus integrity).

Aside from replication for fault tolerance, replication has been explored to detect server penetration [34, 23, 66, 35]. 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., [89, 32]) 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., [32]) addresses multiprocessor VM execution, which our present implementation of *StopWatch* does not. *StopWatch* could be extended to support multiprocessor execution with techniques for deterministic multiprocessor scheduling (e.g., [27]). Mechanisms for enforcing deterministic execution through O/S-level modifications (e.g., [5]) are less relevant to our goals, as they are not easily used by an IaaS cloud provider that accepts arbitrary VMs to execute.

2.2 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., [74, 97]), 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).

2.2.1 Threat Model

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

According to Wray [88], 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 [88]:

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

2.2.2 Defense Strategy

StopWatch is designed to interfere with the use of IO and RT clocks and, for uniprocessor VMs, TL or Mem clocks, for timing attacks. (As discussed in Section 2.1, extension to multiprocessor VMs is a topic of future work.) IO and RT (especially RT) clocks are an ingredient in every timing side-channel attack in the research literature that we have found, undoubtedly because real time is the most intuitive, independent and reliable reference clock for measuring another clock. So, intervening on these clocks is of paramount importance. Moreover, the way *StopWatch* does so forces the scheduler in a uniprocessor guest VM to behave deterministically, interfering with attempts to use TL or Mem clocks.

More specifically, to interfere with IO clocks, *StopWatch* replicates each attacker VM (i.e., every VM, since we do not presume to know which ones are attacker VMs) threefold so that the three replicas of a guest VM are coresident with nonoverlapping sets of (replicas of) other VMs. Then, when determining the timing with which an IO event is made available to each replica, the median timing value of the three is adopted. *StopWatch* addresses RT clocks by replacing a VM's view of real time with a *virtual* time that depends on the VM's own progress, an idea due to Popek and Kline [70].

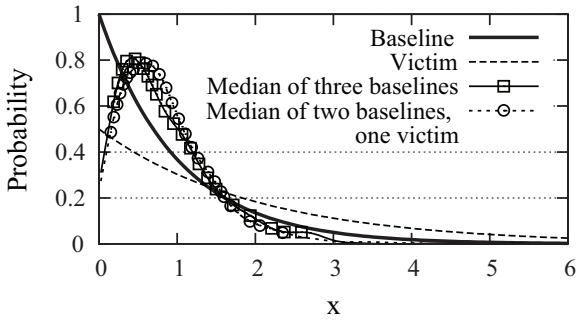
A side effect of how *StopWatch* addresses IO and RT clocks is that it enforces deterministic execution of uniprocessor attacker VM replicas, also disabling its ability to use TL or Mem clocks. These mechanisms thus deal effectively with internal observations of time, but it remains possible that an 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.

2.2.3 Justification for the Median

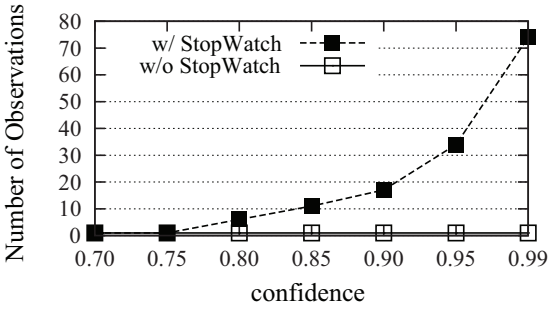
Permitting only the median timing of an IO event to be observed limits the information that an attacker VM can glean from being colocated with a victim VM of interest, because the distribution of the median timings substantially dampens the visibility of a victim's activities.

To see why, consider a victim VM that induces observable timings that are exponentially distributed with rate λ', versus a baseline (i.e., non-victim) exponential distribution with rate $\lambda > \lambda'$.[1] Figure 2.1a 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 Figure 2.1a. In this case, to even reject the null hypothesis that the attacker VM is not coresident with the victim using a χ-square test, the attacker can do so with high confidence in the absence of *StopWatch* with only a single observation, but doing so under *StopWatch* requires almost

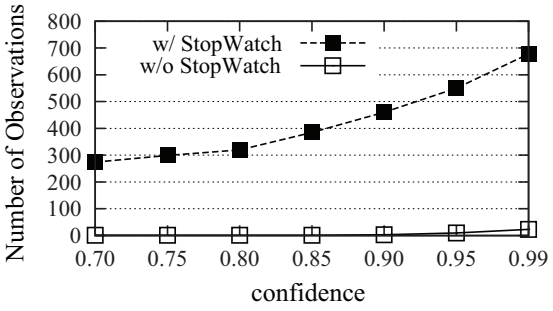
[1] It is not uncommon to model packet inter-arrival time, for example, using an exponential distribution (e.g., [52]).



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



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



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

Figure 2.1: Justification for median; baseline distribution $\mathrm{Exp}(\lambda)$, $\lambda = 1$, and victim distribution $\mathrm{Exp}(\lambda')$

two orders of magnitude more (Figure 2.1b). This improvement becomes even more pronounced if λ and λ' are closer; the case $\lambda = 1$, $\lambda' = 10/11$ is shown in Figure 2.1c.

In terms of the number of observations needed to extract meaningful information from the victim VM, this assessment is very conservative, since the attacker would face numerous pragmatic difficulties that we have not modeled here [97]. But even this simple example shows the power of disclosing only median timings of three VM replicas, and in Section 2.4.2 we will repeat this illustration using actual message traces.

The above illustration of the benefits of allowing only the median timing of an IO event to be observed by an attacker is not specific to timing behaviors that are exponentially distributed. Instead, it generalizes to any distribution. To make this clear, let $X_{r:m}$ denote the random variable that takes on the value of the r-th smallest of the m values obtained by sampling random variables $X_1 \ldots X_m$. Let $F_i(x)$ denote the CDF of X_i (i.e., $F_i(x) = \mathbb{P}(X_i \leq x)$) and let $F_{r:m}(x)$ denote the CDF of $X_{r:m}$. The security of *StopWatch* hinges on the distribution of the median $X_{2:3}$ of three

independent random variables X_1, X_2, X_3 defined as the difference in virtual times (or, in the case of an external observer, real times) between two subsequent IO events.

Specifically, due to the construction of *StopWatch*, the adversary is relegated to learning information from the difference between (i) the CDF $F_{2:3}(x)$ for random variables X_1, X_2, X_3 corresponding to attacker VM replicas that are *not* coresident with a victim VM of interest, and (ii) the CDF $F'_{2:3}(x)$ for random variables X'_1, X_2, X_3 where X'_1 corresponds to an attacker VM that *is* coresident with the victim VM of interest. An example measure of the distance between two CDFs $F(x)$ and $\hat{F}(x)$ is their Kolmogorov-Smirnov distance [28, p. 179], defined as $D(F, \hat{F}) = \max_x \left| F(x) - \hat{F}(x) \right|$.

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

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

Proof. Due to well-known results in order statistics (e.g., see Güngör et al. [41, Result 2.4]):[2]

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

In particular,

$$\begin{aligned} F_{2:3}(x) =& F_1(x)F_2(x) + F_1(x)F_3(x) + F_2(x)F_3(x) - 2F_1(x)F_2(x)F_3(x) \\ F'_{2:3}(x) =& F'_1(x)F_2(x) + F'_1(x)F_3(x) + F_2(x)F_3(x) - 2F'_1(x)F_2(x)F_3(x) \end{aligned}$$

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

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

Noting that $D(F_1, F'_1) = \max_x |F_1(x) - F'_1(x)|$, it suffices to show that $|F_2(x) + F_3(x) - 2F_2(x)F_3(x)| < 1$ for all x. However, since $F_2(x) \in [0, 1]$ and $F_3(x) \in [0, 1]$ for all x, $|F_2(x) + F_3(x) - 2F_2(x)F_3(x)| \leq$

[2] This equation assumes each $F_i(x)$ is continuous. See Güngör et al. [41] for the case when some $F_i(x)$ is not continuous.

1 and, moreover, equals 1 only if for some x, one of $F_2(x)$ and $F_3(x)$ is 1 and the other is 0. This last case is precluded by the theorem. □

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

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

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

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

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

2.3 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).

2.3.1 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 [70]. That is, after the VM executes $instr$ instructions, the virtual time observed from within the VM is

$$virt(instr) \leftarrow slope \times instr + start \tag{2.1}$$

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

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

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

2.3.2 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.

2.3.2.1 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 [87]. 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

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

virtual time $virt$ as computed in Equation 2.1. 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.

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 [84]. 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 [50, Section 29.3.2]. For VM exits caused by guest VM execution, the VMM injects any needed timer interrupts on the next VM entry.

2.3.2.2 `rdtsc` calls and CMOS RTC values

Another way for a guest VM to measure time is via `rdtsc` calls. Xen already emulates the return values to these calls. More specifically, to produce the return value for a `rdtsc` call, the Xen hypervisor computes the time passed since guest reset using its real-time clock, and then this time value is scaled by a constant factor. *StopWatch* replaces this use of a real-time clock with the guest's virtual clock (Equation 2.1).

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.

2.3.2.3 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, instead.

2.4 IO clocks

IO clocks are typically network, disk and DMA interrupts. (Other device interrupts, such as keyboards, mice, graphics cards, etc., are typically not relevant for guest VMs in clouds.) We outline our strategy for mitigating their use to implement timing channels in Section 2.4.1, and then in Section 2.4.2 we describe our implementation of this strategy in *StopWatch*.

2.4.1 Strategy

The method described in Section 2.3 for dealing with RT clocks by introducing virtual time provides a basis for addressing sources of IO clocks. A component of our strategy for doing so is to synchronize I/O events across the three replicas of each guest VM in virtual time, so that every I/O interrupt occurs at the same virtual time at all replicas. Among other things, this synchronization will force uniprocessor VMs to execute deterministically, but it alone will not be enough to interfere with IO clocks; it is also necessary to prevent the timing behavior of one replica's machine from imposing I/O interrupt synchronization points for the others, as discussed in Section 2.1–2.2. 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.

2.4.1.1 Disk and DMA interrupts

The replication of each guest VM at start time includes replicating its entire disk image, and so any disk blocks available to one VM replica will be available to all. By virtue of the fact that (uniprocessor) VMs execute deterministically in *StopWatch*, replicas will issue disk and DMA requests at the same virtual time. Upon receiving such a request from a replica at time V, the VMM adds an offset Δ_d to determine a "delivery time" for the interrupt, i.e., at virtual time $V + \Delta_d$, and initiates the corresponding I/O activities (disk access or DMA transfer). The offset Δ_d must be large enough to ensure that the data transfer completes by the virtual delivery time. Once the virtual delivery time has been determined, the VMM simply waits for the first VM exit caused by the guest VM (as in Section 2.3.2) 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 [49, Sec 4.2.2]).

2.4.1.2 Network interrupts

Unlike the initiation of disk accesses and DMA transfers, the activity giving rise to a network interrupt, namely the arrival of a network packet that is destined for the guest VM, is not synchronized in virtual time across the three replicas of the guest VM. So, the VMMs on the three machines hosting these replicas must coordinate to synchronize the delivery of each network interrupt to the guest VM replicas. To prevent the timing of one from dictating the delivery time at all three, these VMMs exchange proposed delivery times and select the median, as discussed in Section 2.2. To solicit proposed timings from the three, it is necessary, of course, that the VMMs hosting the three replicas all observe each network packet. So, *StopWatch* replicates every network packet to all three computers hosting replicas of the VM for which the packet is intended. This is done by a logically separate "ingress node" that we envision residing on a dedicated computer in the cloud. (Of course, there need not be only one such ingress for the whole cloud.)

When a VMM observes a network packet to be delivered to the guest, it sends its proposed virtual time — i.e., in the guest's virtual time, see Section 2.3 — for the delivery of that interrupt to the VMMs on the other machines hosting replicas of the same guest VM. (We stress that these

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

Once the median proposed virtual time for a network interrupt has been determined at a VMM, the VMM simply waits for the first VM exit caused by the guest VM (as in Section 2.3.2) that occurs at a virtual time at least as large as that median value.[5] The VMM then injects the interrupt prior to the next VM entry of the guest. As with disk accesses and DMA transfers, this interrupt injection also includes copying the data into the address space of the guest, so as to prevent the guest VM from polling for the data in advance of the interrupt to create a form of clock (e.g., [49, Section 4.2.2]).

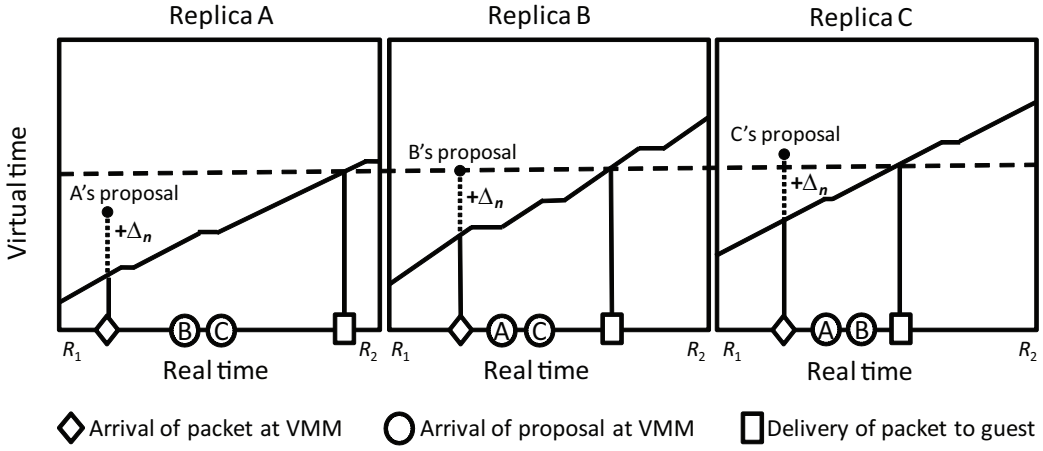


Figure 2.2: Delivering a packet to guest VM replicas.

The process of determining the delivery time of a network packet to a guest VM's replicas is pictured in Figure 2.2. This figure depicts a real-time interval $[R_1, R_2]$ at the three machines at

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

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

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.

2.4.2 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.

2.4.2.1 Network card emulation

In the case of a network card, the device model running in Dom0 receives packets destined for the guest VM. Without *StopWatch* modification, the device model copies this packet to the guest address space and asserts a virtual network device interrupt via the virtual Programmable Interrupt Controller (vPIC) exposed by the VMM for this guest. HVM guests cannot see real external hardware interrupts since the VMM controls the platform's interrupt controllers [50, Section 29.3.2].

In *StopWatch*, we modify the network card device model so as to place each packet destined for the guest VM into a buffer hidden from the guest, rather than delivering it to the guest. The device model then reads the current virtual time of the guest (as of the guest's last VM exit), adds Δ_n to this virtual time to create its proposed delivery (virtual) time for this packet, and multicasts this proposal to the other two replicas (step 1 in Figure 2.3). A memory region shared between Dom0 and the VMM allows device models in Dom0 to read guest virtual time.

Once the network device model receives the two proposals in addition to its own, it takes the median proposal as the delivery time and stores this delivery time in the memory it shares with the VMM. The VMM compares guest virtual time to the delivery time stored in the shared memory upon every guest VM exit caused by guest VM execution. Once guest virtual time has passed the delivery time, the network device model copies the packet into the guest address space (step 2 in Figure 2.3) and asserts a virtual network interrupt on the vPIC prior to the next VM entry (step 3).

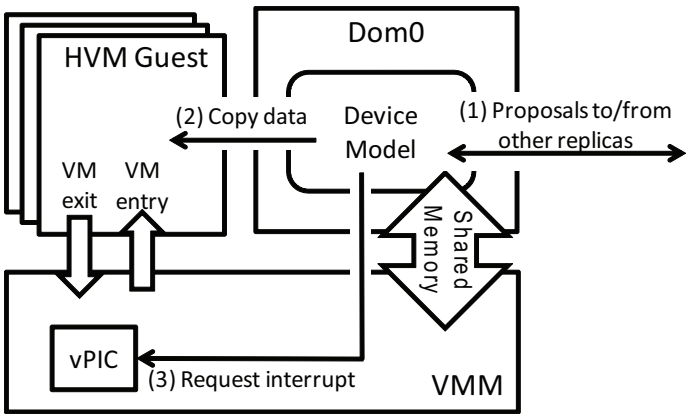
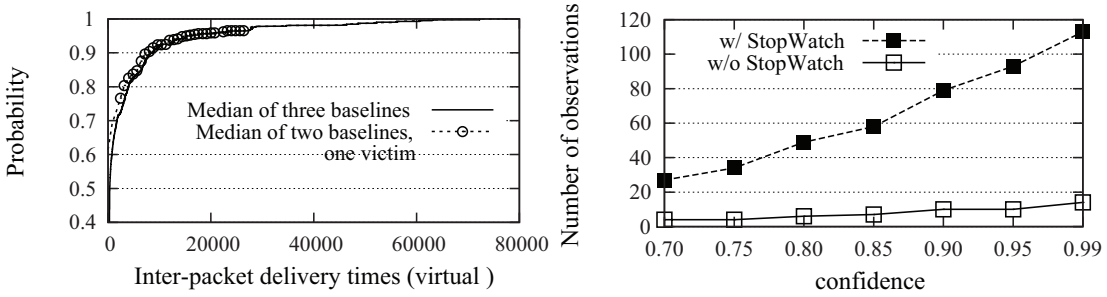


Figure 2.3: Emulation of network I/O device in *StopWatch*.



(a) Distribution of median (CDF)

(b) Observations needed to detect victim

Figure 2.4: Virtual inter-packet delivery times to attacker VM replicas with coresident victim ("two baselines, one victim") and in a run where no replica was coresident with a victim ("three baselines")

Figure 2.4a shows the CDF of virtual inter-packet delivery times to replicas of an attacker VM in an actual run where one replica is coresident with a victim VM continuously serving a file, in comparison to the virtual delivery times with no victim present. This plot is directly analogous to that in Figure 2.1a but is generated from a real *StopWatch* run and shows the distribution as a CDF for ease of readability. Figure 2.4b shows the number of observations needed to distinguish the victim and no-victim distributions in Figure 2.4a using a χ-squared test, as a function of the desired confidence. This figure is analogous to Figure 2.1b 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 [97].

2.4.2.2 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.

2.5 Collaborative Attacks

The mechanisms described in Section 2.3–2.4 intervene on two significant sources of clocks; though VM replicas can measure the progress of one relative to the other, for example, their measurements will be the same and will reflect the median of their timing behaviors. Moreover, by forcing each guest VM to execute (and, in particular, schedule its internal activities) on the basis of virtual time and by synchronizing I/O events across replicas in virtual time, uniprocessor guest VMs execute deterministically, stripping them of the ability to leverage TL and Mem clocks, as well. (More specifically, the progress of TL and Mem clocks are functionally determined by the progress of virtual time and so are not independent of it.) There nevertheless remains the possibility of various collaborative attacks that leverage an attacker VM in conjunction with other attacker components that we discuss below.

2.5.1 External Collaborators

One possible collaborative attack involves conjoining the attacker VM with a collaborator with which it interacts that is external to the cloud and, in particular, on whose real-time clock we cannot

intervene. By interacting with the attacker VM, the external collaborator might attempt to discern information using the real-time behavior of his attacker VM.

Because guest VM replicas will run deterministically, they will output the same network packets in the same order. *StopWatch* uses this property to interfere with a VM's ability to exfiltrate information on the basis of its real-time behavior as seen by an external observer. *StopWatch* does so by adopting the median timing across the three guest VM replicas for each output packet. The median is selected at a separate "egress node" that is dedicated for this purpose (c.f., [90]), analogous to the "ingress node" that replicates every network packet destined to the guest VM to the VM's replicas (see Section 2.4). Like the ingress node, there need not be only one egress node for the whole cloud.

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

An alternative strategy that the external collaborator might take is to send real-time timestamps to his attacker VM, in the hopes of restoring a notion of real-time to that VM (that was stripped away as described in Section 2.3). Again, however, since each packet to the attacker VM is delivered on a schedule dictated by the median progress of the attacker VM replicas (Section 2.4), those timestamps will reflect only on the behavior of the median replica. As such, it matters little whether the external collaborator sends real-time timestamps to the attacker VM or the attacker VM sends virtual-time timestamps (or events reflecting them) to the external collaborator; either way, the power offered by the external collaborator is the same, namely relating progress of the median progress of the attacker VM replicas to real time.

2.5.2 Collaborating Victim-VM Clients

While the type of external collaborator addressed in Section 2.5.1 interacts with the attacker VM, a more powerful collaborator is one that might additionally interact with the victim VM, e.g.,

as one of its clients. This possibility raises the issue of remote timing attacks (e.g., [16]) that do not involve coresidence of attacker VMs with victim VMs at all; such attacks are not our concern here, as we are motivated only by *access-driven* attacks.

That said, recent investigations have paired remote timing attacks with access-driven elements: e.g., Bates et al. [7] and Herzberg et al. [47] developed attacks by which a victim-VM's client could detect the impact of a coresident attacker-VM's communication on the timing of the victim's communication to it, thereby confirming the coresidence of the attacker VM with the victim VM, for example.

While the goal of *StopWatch* is not to defend against all remote timing attacks, it does mitigate the access-driven elements of attacks such as those of Bates et al. and Herzberg et al. Specifically, in *StopWatch* the observable timing of a victim VM's communication to its clients will be dictated by the median progress of its three replicas (Section 2.5.1). As shown in Section 2.2.3, this reveals quantifiably less information to the client than the observable impact of a coresident attacker VM on a (non-replicated) victim VM would. In particular, an attacker VM could perturb the victim VM's observable communication timings only if it is coresident with the victim VM replica whose progress is the median of the victim's three replicas, and only then constrained above and below according to the other replicas' progress.

The defenses suggested by Herzberg et al. to the attack they investigate include a rate-limiting firewall that interferes with the remote attacker's ability to induce load on VMs hosted in the cloud. Our ingress node (Section 2.4.1.2) could trivially be adapted to rate-limit inbound traffic, as well, as a secondary defense against such attacks.

2.5.3 Collaborating Attacker VMs

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

among its replicas' VMMs. The replicas of VM1 would then observe timings influenced by the larger of the replicas on B and C — which may well reflect timings influenced by the victim.

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

If such attacks are nevertheless feared, they can be made harder still by increasing the number of replicas of each VM. If the number were increased from three to, say, five, then inducing sufficient load to marginalize one attacker replica from its median calculations would not substantially increase the attacker's ability to mount attacks on a victim. Rather, the attacker would need to marginalize multiple of its replicas, along with accomplishing the requisite setup to do so.

2.6 Performance Evaluation

In this section we evaluate the performance of our *StopWatch* prototype. We present additional implementation details that impact performance in Section 2.6.1, our experimental setup in Section 2.6.2, and our tests and their results in Section 2.6.3–2.6.4.

2.6.1 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 [81]. In PGM, reliable transmission is accomplished by receivers detecting loss and requesting retransmission of lost data. OpenPGM is used in *StopWatch* for replicating ethernet packets destined to a guest VM to all of

that VM's replicas and for communication among the VMMs hosting guest VM replicas. We also extended the network device model on a host to tunnel each ethernet packet emitted from a local VM replica to the appropriate egress node (see Section 2.5.1) over a persistent TCP connection.

Recall from Section 2.4 that each VMM proposes (via an OpenPGM multicast) a virtual delivery time for each network interrupt, and the VMMs adopt the median proposal as the actual delivery time. As noted there, each VMM generates its proposal by adding a constant offset Δ_n to the current virtual time of the guest VM. Δ_n must be large enough to ensure that by the time each VMM selects the median, that virtual time has not already passed in the guest VM. However, subject to this constraint, Δ_n should be minimized since the real time to which Δ_n translates imposes a lower bound on the latency of the interrupt delivery. (Note that because Δ_n is specified in virtual time and virtual time can vary in its relationship to real time, the exact real time to which Δ_n translates can vary during execution.) We selected Δ_n to accommodate timing differences in the arrivals of packets destined to the guest VM at its three replicas' VMMs, the delays for delivering each VMM's proposed virtual delivery time to the others, and the maximum allowed difference in progress between the two fastest guest VM replicas (which *StopWatch* enforces by slowing the fastest replica, if necessary). For the platform used in our experiments (see Section 2.6.2) and under diverse networking workloads, we found that a value of Δ_n that typically translates to a real-time delay in the vicinity of 7-12ms sufficed to meet the above criteria. The analogous offset Δ_d for determining the virtual delivery time for disk and DMA interrupts was determined based on the maximum observed disk access times and translates to roughly 8-15ms.

2.6.2 Experimental setup

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

provided to the guest as its disk and network card. In each of our tests, we installed an application (e.g., a web server or other program) in the guest VM, as will be described later.

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

Once the guest VM replicas were started, inbound packets for this guest VM were replicated to all three machines for delivery to their replicas as discussed in Section 2.4. These three machines had 100Mb/s ethernet connectivity via a NetGear FS108 switch. They were part of a /24 subnet within the UNC campus network. Broadcast traffic on the network (e.g., ARP requests) was replicated for delivery as in Section 2.4. These broadcasts averaged roughly 50-100 packets per second. As such, this background activity was present throughout our experiments and is reflected in our numbers. Since a cloud operator would presumably place the replicas of each VM in close network proximity to one another so as to minimize the networking penalties of coordinating across those machines, we believe that our doing likewise provides a reasonable approximation of the networking costs that *StopWatch* might encounter in practice.

2.6.3 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 the UNC campus.

2.6.3.1 File downloads

Our first experiments tested the performance of file download by the client from a web server in the cloud. The total times for the client to retrieve files of various sizes over HTTP are shown in Figure 2.5. This figure shows tests in which our guest VM ran Apache version 2.2.14, and the file retrieval was from a cold start (and so file-system caches were empty). The “HTTP Baseline” curve in Figure 2.5 shows the average latency for the client to retrieve a file from an unmodified Xen guest VM. The “HTTP *StopWatch*” curve shows the average cost of file retrieval from our *StopWatch* implementation. Every average is for ten runs. Note that both axes are log-scale.

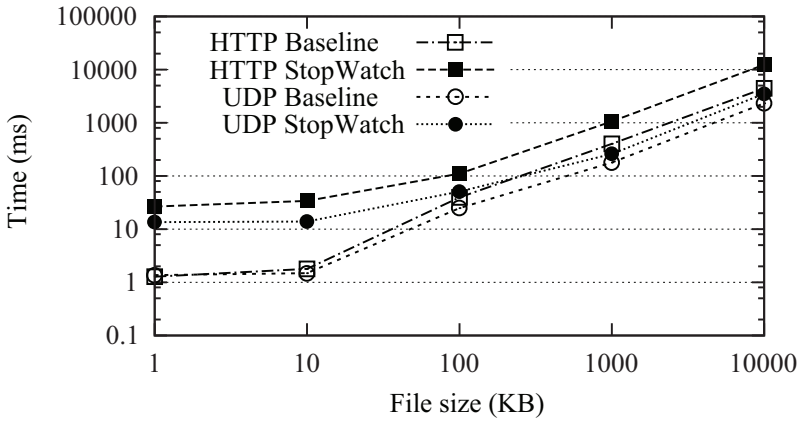
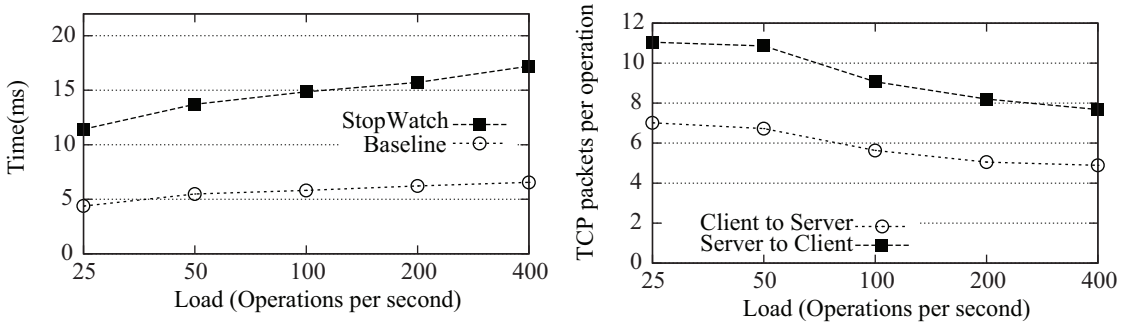


Figure 2.5: HTTP and UDP file-retrieval latency.

Figure 2.5 shows that for HTTP download, a service running on our current *StopWatch* prototype loses less than $2.8\times$ in download speed for files of 100KB or larger. Diagnosing this cost reveals that the bottleneck, by an order of magnitude or more, was the network transmission delay (vs. disk access delay) in both the baseline and for *StopWatch*. Moreover, the performance cost of *StopWatch* in comparison to the baseline was dominated by the time for delivery of *inbound* packets to the web-server guest VM, i.e., the TCP SYN and ACK messages in the three-way handshake, and then additional acknowledgments sent by the client. Enforcing a median timing on output packets (Section 2.5.1) adds modest overhead in comparison.

This combination of insights, namely the detriment of inbound packets (mostly acknowledgments) to *StopWatch* file download performance and the fact that these costs so outweigh disk access costs, raises the possibility of recovering file download performance using a transport protocol that minimizes packets inbound to the web server, e.g., using negative acknowledgments or forward error correction. Alternatively, an unreliable transport protocol with no acknowledgments, such as UDP, could be used; transmission reliability could then be enforced at a layer above UDP using negative acknowledgments or forward error correction. Though TCP does not define negative acknowledgments, transport protocols that implement reliability using them are widely available, particularly for *multicast* where positive acknowledgments can lead to "ack implosion." Indeed, recall that the PGM protocol specification [81], and so the OpenPGM implementation that we use, ensures reliability using negative acknowledgments.



(a) Average latency per op

(b) Average packets per op

Figure 2.6: Tests of NFS server using `nhfsstone`

To illustrate this point, in Figure 2.5 we repeat the experiments using UDP to transfer the file.[6] The "UDP Baseline" curve shows the performance using unmodified Xen; the "UDP *StopWatch*" curve shows the performance using *StopWatch*. Not surprisingly, baseline UDP shows performance comparable to (but slightly more efficient than, by less than a factor of two) baseline TCP, but rather than losing an order of magnitude, UDP over *StopWatch* is *competitive* with these baseline numbers for files of 100KB or more.

2.6.3.2 NFS

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

[6] We are not advocating UDP for file retrieval generally but rather are simply showing the advantages for *StopWatch* of a protocol that minimizes client-to-server packets. We did not use OpenPGM in these tests since the 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 acknowledgments, however.

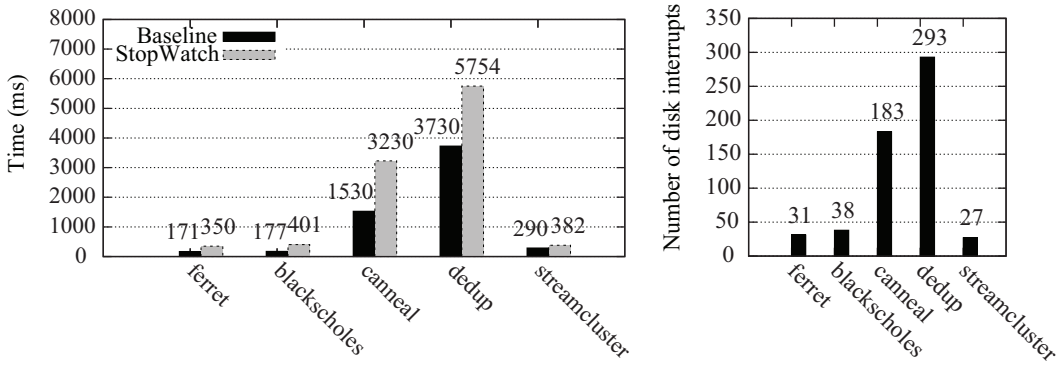
[7] This mix was 11.37% `setattr`, 24.07% `lookup`, 11.92% `write`, 7.93% `getattr`, 32.34% `read` and 12.37% `create`.

The average latency per operation is shown in Figure 2.6a. In this figure, the horizontal axis is the rate at which operations were submitted to the server; note that this axis is log-scale. Figure 2.6a suggests that an NFS server over *StopWatch* incurs a less than $2.7\times$ 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 Figure 2.5. That said, it is also perhaps surprising that *StopWatch*'s cost increased only roughly logarithmically as a function of the offered rate of operations. This modest growth is in part because *StopWatch* schedules packets for delivery to guest VM replicas independently — the scheduling of one does not depend on the delivery of a previous one, and so they can be "pipelined" — and because the number of TCP packets from the client to the server actually decreases per operation, on average, as the offered load grows (Figure 2.6b).

2.6.4 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 [11]. 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.



(a) Average runtimes

(b) Disk interrupts

Figure 2.7: Tests of PARSEC applications

We ran each benchmark ten times in one guest VM over unmodified Xen, and then ten more times with three guest VM replicas over *StopWatch*. Figure 2.7a shows the average runtimes of these applications in both cases. In this figure, each application is described by two bars; the black bar on the left shows its performance over unmodified Xen, and the gray bar on the right shows its performance over *StopWatch*. *StopWatch* imposed an overhead of at most $2.3\times$ (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 mostly due to the overhead involved in intervening on disk I/O (see Section 2.4). As shown in Figure 2.7b, there is a direct correlation between the number of disk interrupts to deliver during the application run and the performance penalty (in absolute terms) that *StopWatch* imposes. If the computers in our experiments used solid-state drives (versus hard disks), we conjecture that their reduced access times would permit us to shrink Δ_d and so improve the performance of *StopWatch* for these applications.

2.7 Comparison to Alternatives

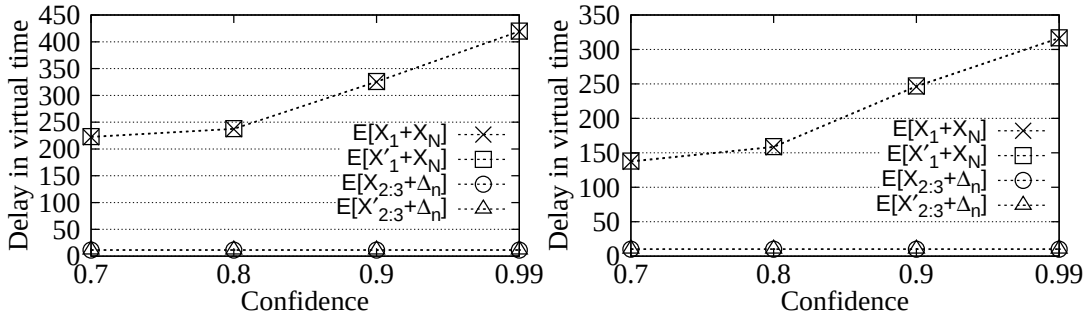
In this section we pause to compare *StopWatch* to two alternatives for defending against timing side-channels of the form we consider here. The two alternatives we consider, neither of which involves VM replication at all, is (i) overcoming timing side-channels by the injection of random noise, and (ii) temporally isolating guest VMs by time slicing each node and running only one guest VM at a time on the node, resetting the machine to as clean a state as possible between each. We discuss these alternatives in Section 2.7.1 and Section 2.7.2, respectively.

The purpose of our comparisons is to illustrate certain advantages that *StopWatch* has over these alternatives, but *not* to argue that *StopWatch* is superior to these alternatives in all ways. Indeed, it is appropriate to point out that *StopWatch*'s approach comes with several deployment overheads that these alternatives do not suffer. For example, *StopWatch* requires VM replication and the placement of each VM's replicas so that the replicas of any VM are coresident with nonoverlapping sets of (replicas of) other VMs, a nontrivial placement constraint discussed further in Section 2.8. Moreover, for any VM for which networking performance is important, the VM replicas should be placed in close network proximity to one another (as we discussed in Section 2.6.2). The cloud must additionally provide (not necessarily physically distinct) ingress nodes for replicating inbound traffic to each VM's replicas (Section 2.4), and egress nodes for hiding timing information in the traffic (the replicas of) each VM sends to others (Section 2.5.1). Neither of the alternatives discussed below impose such additional requirements.

2.7.1 Comparison to Uniformly Random Noise

An alternative to *StopWatch* is simply adding random noise (without replicating VMs) to confound timing attacks. To illustrate advantages that *StopWatch*'s approach has over this alternative, we borrow notation first introduced in Section 2.2.3: Let X_1 denote a random variable representing the "baseline" timing behavior observed by an attacker VM (replica) in the absence of the victim of interest, and let X_1' be the random variable as observed by the attacker VM when it *is* coresident with the victim VM of interest. Again, in *StopWatch*, the adversary learns information from the difference between (i) the distribution of $X_{2:3}$ for random variables X_1, X_2, X_3 corresponding to attacker VM replicas that are *not* coresident with a victim VM of interest, and (ii) the distribution of $X'_{2:3}$ for random variables X_1', X_2, X_3 where X_1' corresponds to an attacker VM that *is* coresident with the victim VM of interest. More specifically, in the case where $X_{2:3}$ or $X'_{2:3}$ denotes the logical time of a network interrupt delivery, for example, the adversary observes either $X_{2:3} + \Delta_n$ or $X'_{2:3} + \Delta_n$. (Δ_n is discussed in Section 2.4.1.2.)

For simplicity, suppose that X_1 and X_1' are exponentially distributed with rate parameters λ and λ', respectively, as in the example of Figure 2.1. For the random variable X_{N} representing added noise, assume that X_{N} is drawn uniformly from $[0, b]$ (i.e., $X_{\mathrm{N}} \sim \mathrm{U}(0, b)$), a common choice to mitigate timing channels (e.g., [49, 39]).



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

Figure 2.8: Expected delay induced by *StopWatch* vs. by uniform noise, as a function of confidence with which attacker distinguishes the two distributions (coresident victim or not) after the same number of observations; baseline distribution $\mathrm{Exp}(\lambda)$, $\lambda = 1$; victim distribution $\mathrm{Exp}(\lambda')$

We calculated expected delay imposed by *StopWatch* and by adding uniformly distributed noise. To make a fair comparison, we configured both approaches to provide the same strength of defense against timing attacks. Specifically, after calculating the number of observations the attacker requires in the case of *StopWatch* to distinguish, for a fixed confidence level, the distributions $X_{2:3} + \Delta_n$ and $X'_{2:3} + \Delta_n$ using a χ-squared test, we calculated the minimum b that would give the attacker the same confidence in distinguishing $X_1 + X_{\mathrm{N}}$ and $X'_1 + X_{\mathrm{N}}$ after that number of observations. Figure 2.8 shows the resulting expected delays in each case.

This figure indicates that *StopWatch* scales much better as the attacker's required confidence and the distinctiveness of the victim grows (as represented by λ' dropping). The delay of the *StopWatch* approach is tied most directly to Δ_n, which is added to ensure that the replicas of each VM remain synchronized (see Section 2.4.1.2); here we calculated it so that $\Pr[|X_1 - X'_1| \leq \Delta_n] \geq 0.9999$. That is, the probability of a desynchronization at this event is less than 0.0001. Note that $E[X_{2:3} + \Delta_n]$ and $E[X'_{2:3} + \Delta_n]$ are nearly the same in Figure 2.8, since their difference is how the attacker differentiates the two, and similarly for $E[X_1 + X_{\mathrm{N}}]$ and $E[X'_1 + X_{\mathrm{N}}]$.

2.7.2 Comparison to Time Slicing

In this section, we compare *StopWatch* to another alternative, namely time slicing, to defend against timing attacks. Here, "time slicing" refers to executing each VM (without replication) in isolation for a period of time. When multiple VMs coreside on the same physical machine, they are scheduled to run in a one-at-a-time fashion. Specifically, time is divided into *slices*, and within

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

Table 2.1: Length of time slice ($sliceLen$) and of cleansing ($cleanseLen$)

each time slice, only one VM is allowed to execute, exclusively occupying all physical resources. VMs are scheduled to consume time slices according to a round-robin scheduler (i.e., in turns). In addition, each two consecutive time slices are separated by a cleansing period within which we cleanse shared components in the system to simulate a machine reset. As an example, Figure 2.9 depicts the execution of three time-sliced VMs running on the same machine.

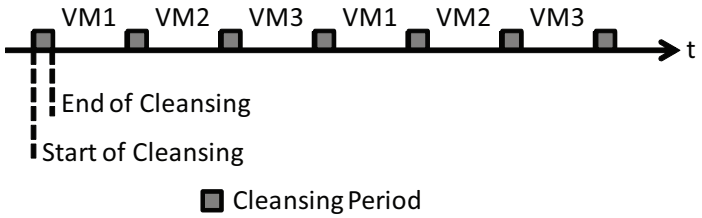


Figure 2.9: Time-sliced execution of three VMs

2.7.2.1 Design

To make VMs execute in turns, we unpause virtual CPUs (vCPUs) of one VM and leave vCPUs of all the other VMs paused for the duration of a time slice. In this experiment, we designed three sets of cleansing operations, described below and summarized in Table 2.1, to flush shared components in the system with varying degrees of thoroughness. Even our most aggressive cleansing operation falls short of a complete machine reset since there are some shared components (e.g., shared network stack of the host machine) with state that could carry information about one VM to another. For each type of cleansing operation described below, we set the length of each time slice ($sliceLen$) to be larger than the length of the cleansing period ($cleanseLen$) by at least one order of magnitude, in an effort to limit the impact of cleansing periods.

Flush-A: CPU caches and TLB In the "Flush-A" cleansing operation, we use the `WBINVD` instruction to flush CPU instruction and data caches. `WBINVD` writes back all modified (instruction and data) cache lines in the processor's internal cache to main memory and invalidates (flushes) the inter-

nal caches. The instruction then directs external caches to be invalidated and to write back modified data, though there are no external caches on our machines in this experiment. In our experiment, `WBINVD` is invoked at the beginning of each cleansing period, as depicted in Figure 2.9.

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

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

Flush-B: Flush-A + Disk page cache The disk page cache is a buffer of disk-backed pages kept in main memory (RAM) by the operating system for quicker access. All physical memory that is not directly allocated to applications is usually used by the operating system for the page cache. For a VM running on Xen, the disk device is virtualized and provided by a device model process running in Dom0. QEMU [8] is used to implement such device models which, by default, uses write-through caching for all block devices (see `http://wiki.qemu.org/download/qemu-doc.html`). This means that the page cache of Dom0 will be used to read and write data. To flush the disk page cache, we use a `SYNC` system call followed by writing to `/proc/sys/vm/drop_caches`. `SYNC` writes all dirty cache pages to the disk, while writing to `/proc/sys/vm/drop_caches` frees all the page caches for reading. In Flush-B, in addition to the CPU and TLB caches, we flush the disk page cache as well, which takes about 185ms in our system. In this case, we set $sliceLen = 2.0$s and $cleanseLen = 0.2$s.

Flush-C: Flush-B + On-drive disk cache buffer The disk cache buffer is the embedded memory in a hard drive acting as a buffer between the rest of the computer and the physical hard disk platter that is used for storage. We use the utility `hdparm -F`, which takes roughly 25ms, to flush this buffer in addition to operations included in Flush-B. In this case, we set $sliceLen = 2.5$s and $cleanseLen = 0.25$s.

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

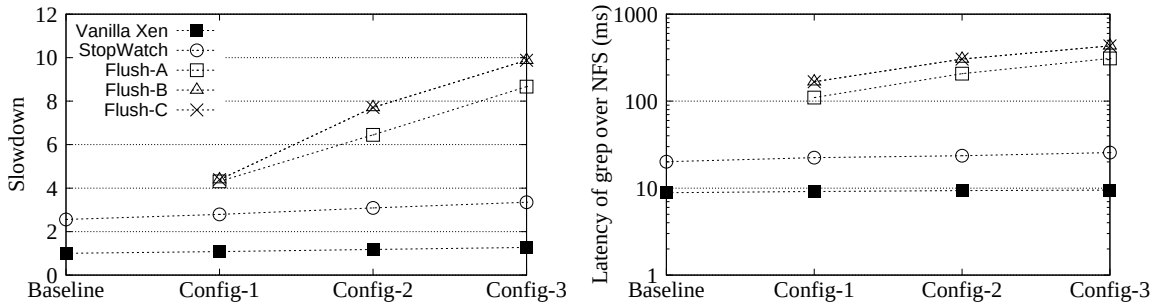
Table 2.2: Configurations

2.7.2.2 Evaluation

To fairly compare the performance of VMs running under *StopWatch* and in a time-slicing fashion, we first configure our system carefully so that the same number of VMs are running on the same number of physical machines in both modes. For instance, given 13 machines, if each is time sliced by two VMs, then there are 26 VMs running in total. *StopWatch* can also support 26 VMs (78 replicas in total) with 13 machines, each of which hosts 6 distinct replicas without violating *StopWatch*'s placement constraints. We have three configurations in this evaluation, shown in Table 2.2, as well as a "Baseline" configuration in which there is at most one VM (replica) per host. In all tests, one "target VM" is serving files via HTTP; half of the other VMs (if any, and rounding up if necessary) with which it is coresident are serving NFS with a workload described below; and the rest are receiving light ICMP traffic (i.e., being `ping`'ed). All VMs in this experiment are uniprocessor VMs. The machines used to support these experiments are as described in Section 2.6.2.

In Figure 2.10a we compare the performance of the target VM serving files via HTTP in the time slicing and *StopWatch* cases. Specifically, the target VM serves a file of size 100MB via HTTP. In these tests, the downloading client was a machine sitting on the same campus network as the nodes hosting these VMs, with a wired connection. The y-axis shows the slowdown factor, which is computed by dividing the time taken to fetch the file from the target VM running in either *StopWatch* or time sliced mode by the "vanilla Xen" value for that configuration. Each shown data point is the average over ten such downloads.

To help explain results shown in Figure 2.10a, in Figure 2.11 we show the progress of downloading for various setups. (Flush-B is not shown, since it largely overlays Flush-C.) Even in Flush-A, the download speed suffers both from frequent context switches among VMs and from CPU cache flushing. While in Flush-C, which has longer time slices, the download speed roughly recovers



(a) File downloading via HTTP (b) `grep` via NFS

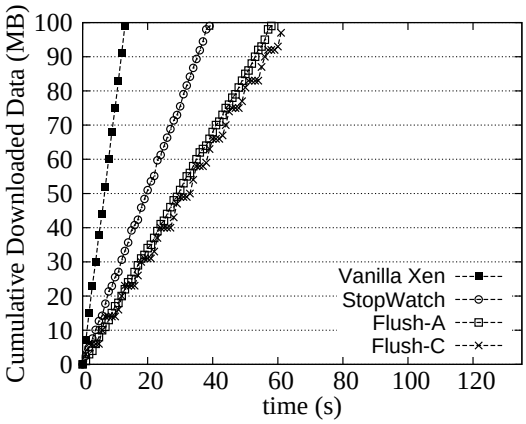
Figure 2.10: *StopWatch* vs. time slicing: comparison of slowdown and delay

within one slice from a cleaned cache, the slice ends shortly thereafter. And also due to the longer time slices, stepped effects become more obvious in Flush-C.

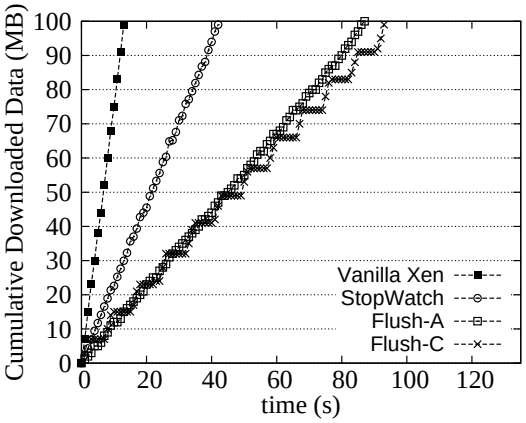
Finally, in Figure 2.10b we confirm these effects by measuring the latency of highly interactive NFS operations. An NFS server was set up in the target VM, and the client remotely mounted the exported partition and then launched `grep` operations, trying to find a target string in a 32B file. `grep` operations were conducted with a frequency of 10 ops/s, and the average latency to perform 200 `grep` operations is reported. In this experiment, the effects observed in the HTTP case manifest themselves as paused `grep` commands owing to the NFS server not being scheduled yet and so being unable to respond.

2.7.3 Discussion

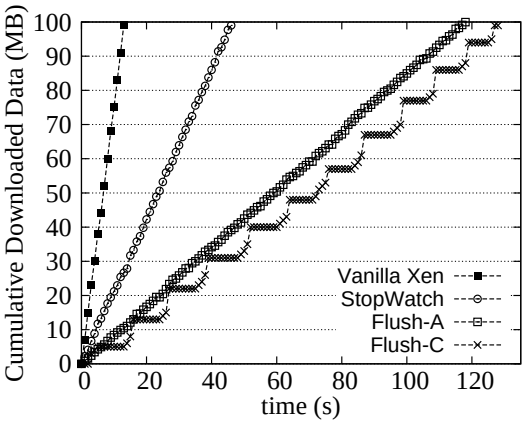
The above analyses are not meant to conclude the *StopWatch* will always provide superior performance to adding random noise or time slicing hosts, nor do we believe that is the case. For example, machines with few physical cores and a compute-intensive, batch workload would almost certainly perform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



(a) Config-1



(b) Config-2



(c) Config-3

Figure 2.11: Progress of file download via HTTP

2.8 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 constrains how a cloud operator places guest VM replicas on

its machines. In this section we clarify the significance of these placement constraints in terms of the provider's ability to best utilize its infrastructure. After all, if under these constraints, the provider were able to simultaneously run a number of guest VMs that scales, say, only linearly in the number of cloud nodes, then the provider should forgo *StopWatch* and simply run each guest VM (non-replicated) in isolation on a separate node. Here we show that the cloud operator is not limited to such poor utilization of its machines. We show some main theorems first and then show their proofs with lemmas.

If the cloud has n machines, then consider the complete, undirected graph (clique) K_n on n vertices, one per machine. For every guest VM, the placement of its three replicas forms a *triangle* in K_n consisting of the vertices for the machines on which the replicas are placed and the edges between those vertices. The placement constraints of *StopWatch* can be expressed by requiring that the triangles representing VM replica placements be pairwise *edge-disjoint*. As such, the number of guest VMs that can simultaneously be run on a cloud of n machines is the same as the number of edge-disjoint triangles that can be *packed* into K_n. A corollary of a result due to Horsley [48, Thm. 1.1] is:

Theorem 3. A maximum packing of K_n with pairwise edge-disjoint triangles has exactly k triangles, where: (i) if n is odd, then k is the largest integer such that $3k \leq \binom{n}{2}$ 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. The existence of such a placement, however, does not guarantee an efficient algorithm to find it. Moreover, this theorem ignores machine capacities. Below we address both of these shortcomings.

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

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

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

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

Next we prove these theorems and some lemmas required to do so.

Let $(\mathbb{Z}_{2v+1}, \oplus)$ denote the cyclic group of addition modulo $2v + 1$ for $v \in \mathbb{N}$, and let $\pi : \mathbb{Z}_{2v+1} \to \mathbb{Z}_{2v+1}$ be a bijection satisfying $\pi(i \oplus i) = i$ for all $i \in \mathbb{Z}_{2v+1}$. (Note that $i \oplus i \neq i' \oplus i'$ for any $i, i' \in \mathbb{Z}_{2v+1}$, $i \neq i'$, since $2v+1$ is odd. As such, π is well defined.) Let $\circledcirc : \mathbb{Z}_{2v+1} \times \mathbb{Z}_{2v+1} \to \mathbb{Z}_{2v+1}$ be defined by $i \circledcirc i' = \pi(i \oplus i')$. Then, $\circledcirc$ is idempotent ($i \circledcirc i = i$ for all $i \in \mathbb{Z}_{2v+1}$) and commutative ($i \circledcirc i' = i' \circledcirc i$ for all $i, i' \in \mathbb{Z}_{2v+1}$). Moreover, for any $i, i' \in \mathbb{Z}_{2v+1}$, $i'' = \pi^{-1}(i') \ominus i$ satisfies $i \circledcirc i'' = i'$, and so $(\mathbb{Z}_{2v+1}, \circledcirc)$ is an idempotent, commutative quasigroup.

Lemma 1. Fix any t, $1 \leq t \leq v$. Then, $\mathbb{Z}_{2v+1} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{i \circledcirc (i \oplus t)\}$.

Proof.

$$\mathbb{Z}_{2v+1} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{i \oplus i\} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{i \oplus i \oplus t\} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{\pi(i \oplus i \oplus t)\} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{i \circledcirc (i \oplus t)\}$$

□

Proof of Theorem 4. Following Bose's construction of a Steiner Triple System [58, Section 1.2], let $n = 6v + 3$ and let $(\mathbb{Z}_{2v+1}, \circledcirc)$ be the idempotent commutative quasigroup of order $2v + 1$ defined above. Let $\mathbb{Z}_{2v+1} \times \{0, 1, 2\}$ denote the n nodes, and consider the following sets G_t, $0 \leq t \leq v$, of triangles:

$$G_0 = \bigcup_{0 \leq i \leq 2v} \{\{(i, 0), (i, 1), (i, 2)\}\}$$

and for $1 \leq t \leq v$,

$$G_t = \bigcup_{\substack{0 \leq i \leq 2v \\ 0 \leq \ell \leq 2}} \{\{(i, \ell), (i \oplus t, \ell), (i \circledcirc (i \oplus t), \ell + 1 \bmod 3)\}\}$$

There are $2v + 1$ triangles in G_0 and $(2v + 1) \times 3 = 6v + 3 = n$ triangles in G_t for each $1 \leq t \leq v$. Moreover, all of these triangles are edge-disjoint [58, Section 1.2]. Triangles in G_0 visit each of the n nodes exactly once. Triangles in any G_t, $1 \leq t \leq v$, visit each node (i^*, ℓ^*) exactly three times: when $i^* = i$ and $\ell^* = \ell$; when $i^* = i \oplus t$ and $\ell^* = \ell$; and when $i^* = i \circledcirc (i \oplus t)$ and $\ell^* = \ell + 1 \bmod 3$. And due to the fact that $(\mathbb{Z}_{2v+1}, \circledcirc)$ is an idempotent, commutative quasigroup, these three times are distinct. Due to Lemma 1, $i \circledcirc (i \oplus t)$ also iterates through the members of $\mathbb{Z}_{2v+1}$ exactly once (i.e., $\mathbb{Z}_{2v+1} = \bigcup_{i \in \mathbb{Z}_{2v+1}} \{i \circledcirc (i \oplus t)\}$). So, collectively the triangles in $G_0, \ldots, G_v$ visit each node $3v + 1 = \frac{n-1}{2} \geq c$ times.

So, if $c \equiv 0 \bmod 3$, then we can place $k \leq \frac{1}{3}cn$ VMs using the $\frac{1}{3}cn$ triangles in groups $G_1, \ldots, G_{c/3}$. If $c \equiv 1 \bmod 3$, then we can place $k \leq \frac{1}{3}cn$ VMs by first using the $2v + 1 = \frac{n}{3}$ triangles in G_0 and then the $\frac{1}{3}(c-1)n$ triangles in $G_1, \ldots, G_{(c-1)/3}$. If $c \equiv 2 \bmod 3$, then we can place $k \leq \frac{1}{3}(c-1)n + \frac{n-3}{6}$ VMs by first using the $2v + 1 = \frac{n}{3}$ triangles in G_0, then $\frac{1}{3}(c-2)n$ triangles in $G_1, \ldots, G_{(c-2)/3}$, and finally any $v = \frac{n-3}{6}$ triangles from G_v that visit each node at most one time (e.g., $\{(i, 0), (i \oplus v, 0), (i \circledcirc (i \oplus v), 1)\}$ for $0 \leq i \leq v - 1$). □

From this point forward, we fix the bijection π to be

$$\pi(i) = \begin{cases} i/2 & \text{if } i \equiv 0 \bmod 2 \\ (i + 2v + 1)/2 & \text{otherwise} \end{cases}$$

Lemma 2. If $2v + 1 \equiv 0 \bmod 3^m$, then:

- If $i \equiv 3^{m-1} \bmod 3^m$, then $\pi(i) \equiv 2 \cdot 3^{m-1} \bmod 3^m$.
- If $i \equiv 2 \cdot 3^{m-1} \bmod 3^m$, then $\pi(i) \equiv 3^{m-1} \bmod 3^m$.

Proof. Since $2v + 1 \equiv 0 \bmod 3^m$ by assumption, we know that $2v + 1 = b \cdot 3^m$ for some $b \in \mathbb{N}$. We first prove if $i \equiv 3^{m-1} \bmod 3^m$, then $\pi(i) \equiv 2 \cdot 3^{m-1} \bmod 3^m$. Note that if $i \equiv 3^{m-1} \bmod 3^m$, then $i = a \cdot 3^m + 3^{m-1}$ for some $a \in \mathbb{N}$.

1. If a is odd and so $a = 2a' + 1$ for some $a' \in \mathbb{N}$, then $i = (2a' + 1)3^m + 3^{m-1} = 2a' \cdot 3^m + 3^m + 3^{m-1} = 2a' \cdot 3^m + (3 + 1)3^{m-1}$. In particular, note that i is even. As a result, $\pi(i) = i/2 = a' \cdot 3^m + 2 \cdot 3^{m-1}$ and so $\pi(i) \equiv 2 \cdot 3^{m-1} \bmod 3^m$.

2. If a is even and so $a = 2a'$ for some $a' \in \mathbb{N}$, then $i = (2a')3^m + 3^{m-1}$, which is odd. Then $\pi(i) = (i + 2v + 1)/2 = (2a' \cdot 3^m + 3^{m-1} + 2v + 1)/2 = a' \cdot 3^m + \frac{3^{m-1}+2v+1}{2} = a' \cdot 3^m + \frac{3^{m-1}+b\cdot 3^m}{2} = a' \cdot 3^m + \frac{3b+1}{2} \cdot 3^{m-1}$. Note that b must be odd, i.e., $b = 2b' + 1$ for some $b' \in \mathbb{N}$. So we have $\pi(i) = a' \cdot 3^m + \frac{6b'+4}{2} \cdot 3^{m-1} = a' \cdot 3^m + 3b' \cdot 3^{m-1} + 2 \cdot 3^{m-1} = (a' + b')3^m + 2 \cdot 3^{m-1}$. So, $\pi(i) \equiv 2 \cdot 3^{m-1} \bmod 3^m$.

Now we prove that if $i \equiv 2 \cdot 3^{m-1} \bmod 3^m$, then $\pi(i) \equiv 3^{m-1} \bmod 3^m$. Note that if $i \equiv 2 \cdot 3^{m-1} \bmod 3^m$, then $i = a \cdot 3^m + 2 \cdot 3^{m-1}$ for some $a \in \mathbb{N}$.

1. If a is even and so $a = 2a'$ for some $a' \in \mathbb{N}$, then $i = (2a')3^m + 2 \cdot 3^{m-1}$. So, $\pi(i) = i/2 = a' \cdot 3^m + 3^{m-1}$ and thus $\pi(i) \equiv 3^{m-1} \bmod 3^m$.

2. If a is odd and so $a = 2a' + 1$ for some $a' \in \mathbb{N}$, then $i = (2a' + 1)3^m + 2 \cdot 3^{m-1}$. Then $\pi(i) = (i + 2v + 1)/2 = ((2a' + 1) \cdot 3^m + 2 \cdot 3^{m-1} + 2v + 1)/2 = a' \cdot 3^m + 3^{m-1} + \frac{3^m+2v+1}{2} = a' \cdot 3^m + 3^{m-1} + \frac{3^m+b\cdot 3^m}{2} = a' \cdot 3^m + 3^{m-1} + \frac{b+1}{2} \cdot 3^m$. Note that b must be odd, i.e., $b = 2b' + 1$ for some $b' \in \mathbb{N}$. So we have $\pi(i) = a' \cdot 3^m + 3^{m-1} + (b' + 1) \cdot 3^n = (a' + b' + 1) \cdot 3^m + 3^{m-1}$ and thus $\pi(i) \equiv 3^{m-1} \bmod 3^m$.

□

Lemma 3. Fix any $i, i' \in \mathbb{Z}_{2v+1}$. Then for any k, $0 \le k \le 2v$, $(i \oplus k) \circledcirc (i' \oplus k) = (i \circledcirc i') \oplus k$.

Proof. We show that for any k, $0 \le k \le 2v$, that $(i \oplus k \oplus 1) \circledcirc (i' \oplus k \oplus 1) \equiv ((i \oplus k) \circledcirc (i' \oplus k)) \oplus 1$. We consider four cases:

- If $(i \oplus k) \oplus (i' \oplus k)$ is even and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1)$ is even, then $\pi((i \oplus k) \oplus (i' \oplus k)) \oplus 1 = \frac{i \oplus k \oplus i' \oplus k}{2} \oplus 1 = \frac{i \oplus k \oplus 1 \oplus i' \oplus k \oplus 1}{2} = \pi((i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1))$.

- If $(i \oplus k) \oplus (i' \oplus k)$ is odd and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1)$ is odd, then $\pi((i \oplus k) \oplus (i' \oplus k)) \oplus 1 = \frac{(i \oplus k) \oplus (i' \oplus k) + 2v + 1}{2} \oplus 1 = \frac{(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1) + 2v + 1}{2} = \pi((i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1))$.

- If $(i \oplus k) \oplus (i' \oplus k)$ is odd and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1)$ is even, then $(i \oplus k) \oplus (i' \oplus k) = 2v - 1$ and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1) = 0$. So, $\pi((i \oplus k) \oplus (i' \oplus k)) \oplus 1 = \frac{2v-1+2v+1}{2} \oplus 1 = 0 = \pi((i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1))$.

- If $(i \oplus k) \oplus (i' \oplus k)$ is even and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1)$ is odd, then $(i \oplus k) \oplus (i' \oplus k) = 2v$ and $(i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1) = 1$. So, $\pi((i \oplus k) \oplus (i' \oplus k)) \oplus 1 = \frac{2v}{2} \oplus 1 = v \oplus 1 = \frac{1+2v+1}{2} = \pi((i \oplus k \oplus 1) \oplus (i' \oplus k \oplus 1))$.

So, for any k, $0 \leq k \leq 2v$, we have that $(i \oplus k \oplus 1) \circledcirc (i' \oplus k \oplus 1) \equiv ((i \oplus k) \circledcirc (i' \oplus k)) \oplus 1$. Therefore, for any k, $0 \leq k \leq 2v$, $(i \oplus k) \circledcirc (i' \oplus k) \equiv (i \circledcirc i') \oplus k$. □

Lemma 4. Let $n = 6v + 3$ and $2v + 1 = 3^q$ for some $q \in \mathbb{N}$. Let $\mathbb{Z}_{2v+1} \times \{0, 1, 2\}$ denote the n nodes, and consider the following sets G_t, $1 \leq t \leq v$, of triangles:

$$G_t = \bigcup_{\substack{0 \leq i \leq 2v \\ 0 \leq \ell \leq 2}} \{\{(i, \ell), (i \oplus t, \ell), (i \circledcirc (i \oplus t), \ell + 1 \bmod 3)\}\}$$

For each G_t, $1 \leq t \leq v$, there exists a set $H_t \subset G_t$ of $2v + 1$ triangles that partition the n nodes.

Proof. To define the subgroup H_t of triangles, we first introduce some variables based on group index t, $1 \leq t \leq v$. Let $m \in \mathbb{N}$ be the maximum value such that $3^{m-1} \mid t$; therefore, $t \equiv 3^{m-1} \bmod 3^m$ or $t \equiv 2 \cdot 3^{m-1} \bmod 3^m$. Then the subgroup of triangles that partition the n nodes is defined as:

$$H_t = \bigcup_{\substack{i \in \mathbb{Z}_{2v+1} \,:\, i \equiv 0 \bmod 3^m, \\ 0 \leq k < 3^{m-1},\, 0 \leq \ell \leq 2}} \{\{(i \oplus k, \ell), (i \oplus t \oplus k, \ell), (((i \oplus k) \circledcirc (i \oplus t \oplus k)), \ell + 1 \bmod 3)\}\} \quad (2.2)$$

To show that (2.2) partitions the nodes, it suffices to show that

$$\mathbb{Z}_{2v+1} = \bigcup_{\substack{i \in \mathbb{Z}_{2v+1} \,:\, i \equiv 0 \bmod 3^m, \\ 0 \leq k < 3^{m-1}}} \{i \oplus k, i \oplus t \oplus k, (i \oplus k) \circledcirc (i \oplus t \oplus k)\} \quad (2.3)$$

Since $2v + 1 = 3^q$, we have $v < 3^q$. And since $t \leq v$, $t < 3^q$. Because $3^{m-1} \mid t$, we have $3^{m-1} \leq t < 3^q$ and so $m \leq q$. Let $p = q - m$. First, we have:

$$\bigcup_{\substack{i\in\mathbb{Z}_{2^{v+1}}\,:\,i\equiv 0 \bmod 3^m,\\ 0\le k<3^{m-1}}} \{i\oplus k\} = \bigcup_{\substack{0\le a<3^p,\\ 0\le k<3^{m-1}}} \{a\cdot 3^m + k\} \tag{2.4}$$

Now suppose t satisfies $t \equiv 3^{m-1} \bmod 3^m$. (The case of $t \equiv 2\cdot 3^{m-1} \bmod 3^m$ is similar.) Denoting $t = b\cdot 3^m + 3^{m-1}$ for some $0 \le b < 3^p$, we have:

$$\begin{aligned}
\bigcup_{\substack{i\in\mathbb{Z}_{2^{v+1}}\,:\,i\equiv 0 \bmod 3^m,\\ 0\le k<3^{m-1}}} \{i\oplus t\oplus k\} &= \bigcup_{\substack{0\le a<3^p,\\ 0\le k<3^{m-1}}} \{(a\cdot 3^m)\oplus(b\cdot 3^m + 3^{m-1})\oplus k\} \\
&= \bigcup_{\substack{0\le a<3^p,\\ 0\le k<3^{m-1}}} \{(a+b \bmod 3^p)\cdot 3^m + 3^{m-1} + k\} \\
&= \bigcup_{\substack{0\le c<3^p,\\ 0\le k<3^{m-1}}} \{c\cdot 3^m + 3^{m-1} + k\} \qquad (2.5)
\end{aligned}$$

Equation 2.5 follows from the fact that $\mathbb{Z}_{3^p} = \{a+b \bmod 3^p\}_{0\le a<3^p}$. Moreover,

$$\begin{aligned}
&\bigcup_{\substack{i\in\mathbb{Z}_{2^{v+1}}\,:\,i\equiv 0 \bmod 3^m,\\ 0\le k<3^{m-1}}} \{(i\oplus k)\circledcirc(i\oplus t\oplus k)\} \\
&= \bigcup_{\substack{i\in\mathbb{Z}_{2^{v+1}}\,:\,i\equiv 0 \bmod 3^m,\\ 0\le k<3^{m-1}}} \{(i\circledcirc(i\oplus t))\oplus k\} \qquad (2.6) \\
&= \bigcup_{\substack{0\le a<3^p,\\ 0\le k<3^{m-1}}} \{\pi((a\cdot 3^m)\oplus(a\cdot 3^m)\oplus(b\cdot 3^m + 3^{m-1})))\oplus k\} \\
&= \bigcup_{\substack{0\le a<3^p,\\ 0\le k<3^{m-1}}} \{\pi(((2a+b \bmod 3^p)\cdot 3^m) + 3^{m-1})\oplus k\} \\
&= \bigcup_{\substack{0\le c<3^p,\\ 0\le k<3^{m-1}}} \{c\cdot 3^m + 2\cdot 3^{m-1} + k\} \qquad (2.7)
\end{aligned}$$

Equation 2.6 follows from Lemma 3 with $i' = i\oplus t$. Equation 2.7 follows from Lemma 2 and the facts that (i) $\mathbb{Z}_{3^p} = \{2a+b \bmod 3^p\}_{0\le a<3^p}$, and (ii) π is a bijection.

Each of the sets in Equation 2.4, Equation 2.5, and Equation 2.7 has size $3^{m-1} \cdot 3^p$, and they clearly do not intersect. So, put together they have size $3^{p+m} = 3^q = 2v + 1$. So, Equation 2.3 holds true, and Equation 2.2 defines $2v + 1$ triangles that exactly partition the n nodes. □

Proof of Theorem 5. Let $(\mathbb{Z}_{2v+1}, \circledcirc)$ be the idempotent commutative quasigroup defined above, and let $\mathbb{Z}_{2v+1} \times \{0, 1, 2\}$ denote the n nodes. Consider the following sets G_t, $0 \leq t \leq v$, of triangles:

$$G_0 = \bigcup_{0 \leq i \leq 2v} \{\{(i, 0), (i, 1), (i, 2)\}\}$$

and for $1 \leq t \leq v$,

$$G_t = \bigcup_{\substack{0 \leq i \leq 2v \\ 0 \leq \ell \leq 2}} \{\{(i, \ell), (i \oplus t, \ell), (i \circledcirc (i \oplus t), \ell + 1 \bmod 3)\}\}$$

As proved in Lemma 4, each G_t, $1 \leq t \leq v$, contains a subset $H_t \subset G_t$ of triangles that partition the n nodes. Let $H_0 = G_0$; this group of triangles also partitions the n nodes. Moreover, note that the triangles in $H_0, \ldots, H_v$ are all edge-disjoint, because all of the triangles in $G_0, \ldots, G_v$ are [58, Section 1.2]. Therefore, $\bigcup_{0 \leq t \leq v} H_t$ contains $(v + 1)(2v + 1)$ edge-disjoint triangles that visit each node $v + 1$ times. VMs can then be placed on any of these triangles that visit each node no more than $\min\{v + 1, c/d_{max}\}$ times. Since $v + 1 = \Theta(n)$ and $\frac{c}{d_{max}} = O(n)$ the number of VMs that can be placed is $\Theta(\frac{c}{d_{max}} n)$. □

2.9 Conclusion

We proposed a new method to address timing side channels in IaaS compute clouds that employs three-way replication of guest VMs and placement of these VM replicas so that they are coresident with nonoverlapping sets of (replicas of) other VMs. By permitting these replicas to observe only virtual (vs. real) time and the median timing of network events across the three replicas, we suppress their ability to glean information from a victim VM with which one is coresident. We described an implementation of this technique in Xen, yielding a system called *StopWatch*, and we evaluated the performance of *StopWatch* on a variety of workloads. Though the performance cost for our current prototype ranges up to $2.8\times$ for networking applications, we used our evaluation

to identify the sources of costs and alternative application designs (e.g., reliable transmission using negative acknowledgments, to support serving files) that can enhance performance considerably. We also extended this evaluation to demonstrate workloads for which *StopWatch* provides better performance than alternatives that leverage commodity hardware, namely adding random noise to observable event timings and eliminating concurrent VM execution (time slicing). We showed that clouds with n machines capable of each running $c \leq \frac{n-1}{2}$ guest VMs simultaneously can efficiently schedule $\Theta(cn)$ guest VMs under the constraints of *StopWatch*, or $\Theta(\frac{cn}{d_{max}})$ guest VMs if each guest VM makes demands on the per-machine capacity c of at most d_{max}. These results represent a clear improvement over the alternative of running each guest VMs on its own machine. We envision *StopWatch* as a basis for a high-security cloud, e.g., suitable for military, intelligence, or financial communities with high assurance needs.

An important topic for future work is extending *StopWatch* to support multiprocessor guest VMs. As discussed in Section 2.1, previous research on deterministic scheduling (e.g., [27]) should provide a basis for extending our current *StopWatch* prototype. A second direction for improvement is that we have implicitly assumed in our *StopWatch* implementation — and in many of our descriptions in this chapter — that the replicas of each VM are placed on a set of homogeneous nodes. Expanding our implementation to heterogeneous nodes poses additional challenges that we hope to address in future work.

CHAPTER 3 REPLICA PLACEMENT FOR AVAILABILITY IN THE WORST CASE

In this chapter, we consider the problem of deploying *replicas* of *objects* onto a system of physical *nodes* so as to ensure the survival of as many objects as possible when node failures occur. This general problem occurs in practice in many computing contexts: the "objects" might be virtual machines, files, or servers, and the "replicas" could be whole object copies or merely components used in the implementation of the object. The survival of an object is achieved provided that fewer than a given threshold number of its replicas were placed on the nodes that fail. This threshold might range from all of the object replicas to only a few. The question we address in this chapter is: How should the object replicas be placed on the nodes (aside from the obvious requirement that the replicas of an object all be placed on different nodes)?

Upon encountering this problem for the first time, it might not be immediately obvious that the placement matters. But consider the possibility that all of the failed nodes host replicas of mostly the same objects. This scenario might fail objects that require many replica failures to do so, but it fails fewer objects than it otherwise could if each object fails when only a few of its replicas do. Alternatively, suppose the failed nodes host replicas of mostly different objects. Then, many objects might fail if only few object replica failures suffice to fail each object, but fewer objects might fail if many replica failures per object are required. As this contrast suggests, the placement certainly matters and depends not only on the number of nodes, the number of objects, the number of node faults, and the replicas per object, but also the number of an object's replicas' failures that prove fatal to the object.

We are not the first to study the problem of object replica placement for availability (see Section 3.1 for a discussion of related work), but to our knowledge, our treatment is novel in at least two ways. First, we consider a *worst-case* adversary that fails a specified number of nodes *with knowledge of how object replicas were placed on nodes*, so as to maximize the number of objects failed. This is in contrast to failures that occur probabilistically, for example. Second, by decoupling

the number of replicas per object from the number of replica failures that disable each object, our framework allows for treatment of a wide variety of object configurations, such as objects that are accessed using majority quorums (e.g., [38, 36]) so that a majority of available replicas is required for the object to survive, or objects for which even just a single surviving replica suffices to keep the object available (e.g., in the primary-backup(s) approach [17]).

Our study is also general by virtue of what it leaves unspecified. While we label nodes, replicas and objects as "failed" or not, we remain agnostic to the fault model [79] (crash, Byzantine, etc.). Indeed, our interest in this problem arose from our work for using virtual machine replication as a defense against timing side-channels in an infrastructure-as-a-service compute cloud [57] (detailed in Chapter 2), without attention to actual faults at all. Similarly, the protocols run among object replicas or for objects to interact with others are not our concern here. Rather, we simply assume that a node failure fails all of the object replicas it hosts, and that an object fails once a specified number of its object replicas do. We also do not constrain the means by which the adversary fails the nodes it chooses to, whether that be disabling them by denial-of-service attacks, leveraging vulnerabilities in object replicas they host, physically attacking the nodes, etc.

In this context, we make the following contributions:

- We study the viability of block designs for replica placements. Specifically, we first leverage t-packings (e.g., see [61]), a relaxation of Steiner systems, as a replica placement strategy. We provide a lower bound for the availability of these replica placements and show that they already offer availability that is c-competitive with optimal placements (for a factor c that we specify). This suggests that t-packings are a useful starting point for constructing placement strategies.
- We develop a placement strategy that improves on the use of t-packings in isolation by combining them. We present an efficient algorithm to compute combinations of individual t-packings that maximize our lower bound on availability (among any such combination) for a given number of node failures. We further demonstrate that for a range of practical parameter values, the placement strategy derived for a given target number of failures provides good availability even for different numbers of failures.
- We develop as our primary comparison point a placement strategy of randomly placing replicas on nodes subject to a load-balancing requirement, owing to the popularity of this strategy in

previous work. We characterize availability for this placement strategy in our adversarial model, and then we develop an expression for the limit of this measure as the number of objects grows. We further show that this limit already closely reflects reality for practical parameter values and relatively small numbers of objects, allowing it to be used as a basis to compare to the availabilities offered by our strategies based on t-packings. In this way, we show that our constructions based on t-packings provide better availability for ranges of practical parameter values than does random replica placement.

As discussed above, the replica placement strategies that we explore build from t-packings, and some of our analysis depends on use of *maximum* t-packings (also called t-designs). Based on current knowledge of t-designs (which we briefly survey in Section 3.2.3), we limit our attention to replication scenarios involving up to five replicas per object. Fortunately, this decision is not limiting for practical replication scenarios in data centers: VM replication for fault tolerance typically uses two (e.g., [86]) and many file systems default to three or four replicas per file or related structure (as in GFS [37], Hadoop [80], and FARSITE [2]).

3.1 Related Work

Replica placement for availability (or durability) has been extensively studied in various fields (e.g., [12, 31, 10, 93, 68, 91, 9, 75]), sometimes in conjunction with other concerns. All related work we have located focuses on leveraging node failure distributions, especially their independence and/or heterogeneity as would be common in peer-to-peer storage and computing, for example. Here we make no assumptions about node failure distributions, allowing them to be controlled by an arbitrary adversary constrained only by the number of nodes he can fail. This renders our analysis both simpler in many cases and, at the same time, very general.

We nevertheless draw from this work where possible. Notably, at PODC 2007, Yu and Gibbons [91] explored the following question: If each node fails independently with fixed probability and if all replicas of an object must fail for the object to fail, then what placement strategy offers the highest probability of success for operations involving multiple objects, a given number of which must be available for the operation to succeed? Their finding that we most directly leverage here is their identification of random replica placements as offering close to the best probability of opera-

tion success when an operation can tolerate some object failures. Together with the widespread use and empirical study of random placements (e.g., [76, 2, 10, 37, 92, 59]), this finding motivates our choice of random replica placement as a comparison point for our proposed placement strategies in Section 3.3. That said, for drawing this comparison we need to develop our own analysis of the availability of random placements, since we focus on a worst-case adversary that can choose which nodes to fail; this analysis might be of interest in its own right. Our work also differs from Yu and Gibbons' in that we do not consider multi-object operations, asking instead only how many objects remain available, but we do so while permitting an object to remain available only if a specified number of its replicas survive (versus just one of them). Note that equating our "objects" to their "operations" and our "replicas" to their "objects" (each with only one replica) does not yield the same problem—even setting aside our different adversarial models—since replicas of the same object in our case must be placed on different nodes, while their objects do not.

As discussed earlier, the cornerstone of the replica placement strategy we develop is a t-packing. To our knowledge, we are the first to explore the use of t-packings for replica placement in distributed systems. That said, such block designs have found application in several diverse domains, as surveyed elsewhere (e.g., [22, 20, 71]). To our knowledge, the most conceptually related use of block designs to our problem is their use in constructing quorum systems (e.g., [65]). Quorum systems, however, must intersect, whereas we have no such requirement here for object placements, a fact that we leverage.

3.2 Overlap-Based Placement Strategies

The strength of random replica placement in diminishing the likelihood that random node failures will fail many objects (see Section 3.1) derives from it inducing low *inter-object correlation* [91], a measure that reflects the overlaps of objects' replica placements. However, random placement induces small overlaps only probabilistically, allowing the possibility that *targeted* node failures could still impact many objects. In this section we explore "overlap-based" placement strategies that manage these overlaps explicitly. We will return to analyzing the impact of targeted node failures on random placements in Section 3.3 and compare to our overlap-based strategies there.

b	The number of objects
r	The number of replicas per object
s	The number of an object's replicas whose failure fails the object; $1 \leq s \leq r$
n	The number of nodes
k	The number of failed nodes; $s \leq k < n$
π	A placement
$\mathcal{O}$	The set of all objects; $\lvert\mathcal{O}\rvert = b$
$\mathcal{N}$	The set of all nodes; $\lvert\mathcal{N}\rvert = n$

Figure 3.1: Notation

Before continuing, we first define some notation used in the rest of this document (see Figure 3.1). We presume a system of n nodes denoted by the set $\mathcal{N}$ ($|\mathcal{N}| = n$). These nodes will host a set $\mathcal{O}$ of b objects ($|\mathcal{O}| = b$), each replicated r times. This hosting is represented by a *placement* $\pi : \mathcal{O} \to \mathbf{2}^{\mathcal{N}}$, where $\mathbf{2}^{\mathcal{N}}$ is the power set of $\mathcal{N}$. Specifically, for each $\mathsf{obj} \in \mathcal{O}$, $\pi(\mathsf{obj})$ is a subset of $\mathcal{N}$ of size $|\pi(\mathsf{obj})| = r$ that indicates the nodes on which replicas of obj are located. We use k to denote the number of nodes that fail. If $\mathcal{K} \subseteq \mathcal{N}$ is the set of k failed nodes, then an object obj is said to fail if and only if $|\pi(\mathsf{obj}) \cap \mathcal{K}| \geq s$. This gives rise to the following natural definition of the availability of a placement π.

Definition 1. For any fixed placement π, let $\mathit{Avail}(\pi)$ denote the number of available objects, minimized over all sets $\mathcal{K}$ of (potentially failed) nodes where $|\mathcal{K}| = k$. In other words,

$$\mathit{Avail}(\pi) = \min_{\substack{\mathcal{K} \subseteq \mathcal{N}: \\ |\mathcal{K}| = k}} |\{\mathsf{obj} \in \mathcal{O} : |\pi(\mathsf{obj}) \cap \mathcal{K}| < s\}|$$

3.2.1 The $\mathsf{SimpleOverlap}(x, \lambda)$ Placement Strategy

Our intuition for developing a replica placement strategy so as to maximize availability is simply to limit the number of objects whose replicas overlap on the same nodes "too much." This intuition is captured in the $\mathsf{SimpleOverlap}(x, \lambda)$ strategy, which limits overlaps of more than x nodes to at most λ objects. We limit our attention to $x < s$, since once $x \geq s$, arbitrarily many objects can overlap on s nodes in a $\mathsf{SimpleOverlap}(x, \lambda)$ placement, meaning that failures of those nodes could fail arbitrarily many objects.

Definition 2. The SimpleOverlap(x, λ) placement strategy locates object replicas on nodes so that for all $\mathcal{N}' \subseteq \mathcal{N}$ where $|\mathcal{N}'| = x + 1$ and all $\mathcal{O}' \subseteq \mathcal{O}$, if on every node in $\mathcal{N}'$ is placed replicas of all objects in $\mathcal{O}'$, then $|\mathcal{O}'| \leq \lambda$.

So, for example, if $\lambda = 1$, then the replicas of any two objects can overlap on at most x nodes.

It is important to note that a SimpleOverlap(x, λ) placement exists only for limited values of b, once n and r are fixed. Specifically, from design theory results, where a SimpleOverlap(x, λ) is otherwise known as a $(x + 1)$-(n, r, λ)-*packing* (e.g., [61]), we have:

Lemma 5 e.g., [61]. A SimpleOverlap(x, λ) placement exists only if $b \leq \left\lfloor \lambda \binom{n}{x+1} / \binom{r}{x+1} \right\rfloor$.

While $b \leq \left\lfloor \lambda \binom{n}{x+1} / \binom{r}{x+1} \right\rfloor$ is necessary for a SimpleOverlap(x, λ) placement, it is not sufficient. To achieve a sufficient condition, we select an $n_x \leq n$ and a μ_x of which λ is a multiple (i.e., $\mu_x \mid \lambda$), as a function of x (and r, which we generally consider a constant) so that $\mu_x \binom{n_x}{x+1} / \binom{r}{x+1}$ is integral and, moreover, a SimpleOverlap(x, μ_x) placement exists for any $b \leq \mu_x \binom{n_x}{x+1} / \binom{r}{x+1}$ objects. Then, a SimpleOverlap(x, λ) placement on n_x nodes can be obtained by "copying" the SimpleOverlap(x, μ_x) placement λ/μ_x times.

Observation 1. If there exist an $n_x \leq n$ and a $\mu_x \mid \lambda$ so that a SimpleOverlap(x, μ_x) placement exists for all $b \leq \mu_x \binom{n_x}{x+1} / \binom{r}{x+1}$, then a SimpleOverlap$(x, \lambda)$ placement exists for all $b \leq \lambda \binom{n_x}{x+1} / \binom{r}{x+1}$.

Observation 2. Placing replicas on only $n_x \leq n$ nodes can lead to a load-imbalanced system, but only slightly if we can find a suitable $n_x \approx n$. If we cannot, then we can instead identify values n_{x1}, ..., n_{xm} such that $\sum_{i=1}^{m} n_{xi} \leq n$ but $\sum_{i=1}^{m} n_{xi} \approx n$, and then extend the results below to account for building a SimpleOverlap(x, λ) placement on $\sum_{i=1}^{m} n_{xi}$ nodes for any $b \leq \sum_{i=1}^{m} \lambda \binom{n_{xi}}{x+1} / \binom{r}{x+1}$ objects from a SimpleOverlap(x, λ) placement on each chunk of n_{xi} nodes.

The extension in Observation 2 is straightforward but tedious, and so we defer its discussion to Section 3.2.3. For now, we simply assume that a suitable n_x and μ_x exist and can be found to support Observation 1. We also adopt the convention that, given n_x, μ_x, r, s, and b, λ is chosen minimally, so that

$$(\lambda - \mu_x) \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}} < b \leq \lambda \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}} \tag{3.1}$$

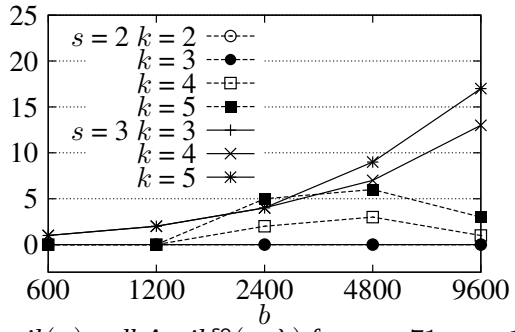


Figure 3.2: $\mathit{Avail}(\pi) - \mathit{lbAvail}^{\mathsf{so}}(x, \lambda)$ for $n = 71$, $x = 1$, and $r = 3$

We now briefly characterize the availability of $\mathsf{SimpleOverlap}(x, \lambda)$ placements, to justify their use as a building block for a more useful placement strategy in Section 3.2.2. The key observation in characterizing the availability of $\mathsf{SimpleOverlap}(x, \lambda)$ placements is that the availability can be lower-bounded by applying Lemma 5 to packing s-sized sets of replicas into the k failed nodes, as shown in the following lemma.

Lemma 6. For any $\mathsf{SimpleOverlap}(x, \lambda)$ placement π, $\mathit{Avail}(\pi) \geq \mathit{lbAvail}^{\mathsf{so}}(x, \lambda)$ where

$$\mathit{lbAvail}^{\mathsf{so}}(x, \lambda) = b - \left\lfloor \lambda \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor \tag{3.2}$$

Proof. An upper bound on the number of objects that become unavailable due to the failure of nodes in $\mathcal{K}$ is simply the number of objects for which s replicas can be packed onto the nodes $\mathcal{K}$ under the constraints of a $\mathsf{SimpleOverlap}(x, \lambda)$ placement, i.e., in a $\mathsf{SimpleOverlap}(x, \lambda)$ placement using only s replicas per object (versus r) and only k nodes (versus n). Adapting Lemma 5 accordingly, we get that at most $\left\lfloor \lambda \binom{k}{x+1} / \binom{s}{x+1} \right\rfloor$ objects become unavailable. □

$\mathit{lbAvail}^{\mathsf{so}}(x, \lambda)$ is a tight lower bound for some but not all parameter values, as indicated in Figure 3.2. In this figure, $\mathit{Avail}(\pi)$ was calculated explicitly after placing objects according to a $\mathsf{SimpleOverlap}(x, \lambda)$ placement π and then simulating the worst k failures.

This lower bound for $\mathit{Avail}(\pi)$, together with Equation 3.1, permits us to relate $\mathit{Avail}(\pi)$ to the availability of *any* placement π'—and so, in particular, one offering optimal availability.

Theorem 6. For constant n, x (and so n_x, μ_x), r, s, and k, define constants $c = \left[1 - \frac{\binom{r}{x+1}\binom{k}{x+1}}{\binom{n_x}{x+1}\binom{s}{x+1}}\right]^{-1}$ and $\alpha = c\mu_x \frac{\binom{k}{x+1}}{\binom{s}{x+1}}$. For any number b of objects, any $\mathsf{SimpleOverlap}(x, \lambda)$ placement π, and any

other placement π',

$$\mathit{Avail}(\pi') < c \cdot \mathit{Avail}(\pi) + \alpha$$

In this respect, $\mathsf{SimpleOverlap}(x, \lambda)$ placements are "c-competitive" (c.f., [14]) with optimal placements.

Proof. First note that Equation 3.1 implies

$$\frac{\lambda}{b} < \frac{\binom{r}{x+1}}{\binom{n_x}{x+1}} + \frac{\mu_x}{b} \tag{3.3}$$

By Lemma 6,

$$\frac{\mathit{Avail}(\pi)}{\mathit{Avail}(\pi')} \geq \frac{b - \left\lfloor \lambda \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor}{b} \geq \frac{b - \lambda \frac{\binom{k}{x+1}}{\binom{s}{x+1}}}{b} = 1 - \frac{\lambda}{b} \frac{\binom{k}{x+1}}{\binom{s}{x+1}} > 1 - \left(\frac{\binom{r}{x+1}}{\binom{n_x}{x+1}} + \frac{\mu_x}{b} \right) \left(\frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right)$$

where the last step is simply substituting Equation 3.3. Rearranging, we get

$$\mathit{Avail}(\pi') - c \cdot \mathit{Avail}(\pi) < \left(\frac{\mathit{Avail}(\pi')}{b} \right) \alpha \leq \alpha$$

where c and α are as given in the theorem statement. □

To see an illustration of Theorem 6, suppose that $s = r$ so that $\binom{r}{x+1}$ and $\binom{s}{x+1}$ cancel. Then,

$$c = \left[1 - \frac{\binom{r}{x+1}\binom{k}{x+1}}{\binom{n_x}{x+1}\binom{s}{x+1}} \right]^{-1} = \left[1 - \frac{k(k-1)\cdots(k-x)}{n_x(n_x-1)\cdots(n_x-x)} \right]^{-1} \leq \left[1 - \left(\frac{k}{n_x} \right)^{x+1} \right]^{-1}$$

So, for example, if $\left(\frac{k}{n_x} \right)^{x+1} = 0.2$, then the availability of a $\mathsf{SimpleOverlap}(x, \lambda)$ placement is 1.25-competitive with the availability offered by an optimal placement. On the other hand, under other conditions (such as when s is small relative to r), this constant factor can be less favorable.

3.2.2 The $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ Placement Strategy

The previous section illustrated the potential utility of $\mathsf{SimpleOverlap}(x, \lambda)$ placements, but we stopped short of suggesting exactly how to select x. To see why this may not be straightforward, consider a fixed n, r, s, and k, but consider increasingly large values of b. On the one hand, if x is

held constant, then the value λ must grow linearly with b, due to Equation 3.1. This, however, implies that the (lower bound on) availability in Lemma 6 also *diminishes* linearly. On the other hand, if x is increased so that λ need not be, then this increases the values of b that can be accommodated exponentially (assuming each $n_x \approx n$ and $r \ll n$); to accommodate some values of b, though, this huge increase is unnecessary and results in a larger penalty to availability than increasing λ would have.

In this section we develop a new placement strategy, called $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$, that provides us the flexibility to tune parameters $\lambda_0, \ldots, \lambda_{s-1}$ corresponding to the possible values of x, $0 \leq x < s$, to best match a given b. That is, $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ takes a value λ_x corresponding to each x, $0 \leq x < s$, subject to the constraint

$$b \leq \sum_{x=0}^{s-1} \lambda_x \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}} \tag{3.4}$$

and then divides the objects over placements $\mathsf{SimpleOverlap}(0, \lambda_0), \ldots, \mathsf{SimpleOverlap}(s-1, \lambda_{s-1})$. Equation 3.4 ensures that $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ can accommodate all b objects, since each $\mathsf{SimpleOverlap}(x, \lambda_x)$ placement can accommodate $\lambda_x \binom{n_x}{x+1} / \binom{r}{x+1}$ of them (see Observation 1).

Definition 3. A $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement strategy locates object replicas on nodes by placing up to $\lambda_x \binom{n_x}{x+1} / \binom{r}{x+1}$ objects according to a $\mathsf{SimpleOverlap}(x, \lambda_x)$ placement for each $x \geq 0$.

Lemma 7. For any $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement π, $\mathit{Avail}(\pi) \geq \mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ where

$$\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) = b - \sum_{x=0}^{s-1} \left\lfloor \lambda_x \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor \tag{3.5}$$

Proof. Under a $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement, each $\mathsf{SimpleOverlap}(x, \lambda_x)$ placement accounts for placing at most $\lambda_x \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}}$ objects, of which up to $\left\lfloor \lambda_x \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor$ might be rendered unavailable by k node failures, as in Lemma 6. As such, at most $\sum_{x=0}^{s-1} \left\lfloor \lambda_x \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor$ objects can be rendered unavailable in total by k node failures. Since only b objects can be placed, the result follows. $\square$

3.2.2.1 Computing a $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ to Maximize *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$

To maximize availability using $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ for a given value of k, we thus take it as our goal to select $\lambda_0, \ldots, \lambda_{s-1}$ so as to maximize the lower bound *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ subject to Equation 3.4. This problem lends itself to the following recurrence for $\mathsf{lbav}\,(x, b')$, which denotes this maximum value of *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ for b' objects placed using placements $\mathsf{SimpleOverlap}(0, \lambda_0)$, ..., $\mathsf{SimpleOverlap}(x, \lambda_x)$ under any selection of $\lambda_0, \ldots, \lambda_x$.

$$\forall x, \forall b' \leq 0 : \ \mathsf{lbav}\left(x, b'\right) = 0 \tag{3.6}$$

$$\forall b' > 0 : \ \mathsf{lbav}\left(0, b'\right) = \max\left\{0, b' - \left\lfloor \left(\left\lceil \frac{b'}{\mu_0} \frac{r}{n_0} \right\rceil \mu_0 \right) \frac{k}{s} \right\rfloor \right\} \tag{3.7}$$

$$\forall x > 0, \forall b' > 0 : \ \mathsf{lbav}\left(x, b'\right) = \max_{0 \leq d \leq \left\lceil \frac{b'}{\mu_x} \frac{\binom{r}{x+1}}{\binom{n_x}{x+1}} \right\rceil} \left\{ \mathsf{lbav}\left(x - 1, b' - d\mu_x \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}} \right) + \min\left\{ b', d\mu_x \frac{\binom{n_x}{x+1}}{\binom{r}{x+1}} \right\} - \left\lfloor d\mu_x \frac{\binom{k}{x+1}}{\binom{s}{x+1}} \right\rfloor \right\} \tag{3.8}$$

In words, Equation 3.6 encodes that zero availability can be offered if there are no objects ($b' \leq 0$). Equation 3.7 encodes that when $x = 0$ the availability that can be achieved for $b' > 0$ objects is that resulting from setting $\lambda_0 = \left\lceil (b'/\mu_0)\binom{r}{1}/\binom{n_0}{1} \right\rceil \mu_0 = \left\lceil (b'/\mu_0)(r/n_0) \right\rceil \mu_0$ and using Lemma 6 (or simply 0 if this value turns out to be negative). Finally, Equation 3.8 encodes that when $x > 0$, availability can be maximized by considering every option for $\lambda_x = d\mu_x$ and, for each option, adding the availability contributed by this setting of λ_x (i.e., $\min\left\{b', \lambda_x \binom{n_x}{x+1}/\binom{r}{x+1}\right\} - \left\lfloor \lambda_x \binom{k}{x+1}/\binom{s}{x+1} \right\rfloor$) to the availability that can be achieved for the remaining $b' - \lambda_x \binom{n_x}{x+1}/\binom{r}{x+1}$ objects by optimally setting $\lambda_1, \ldots, \lambda_{x-1}$ (i.e., $\mathsf{lbav}\left(x - 1, b' - \lambda_x \binom{n_x}{x+1}/\binom{r}{x+1}\right)$).

So, $\mathsf{lbav}\,(s-1, b)$ for a given number k of failed nodes is the maximum *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ that a $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement π can achieve. This recurrence gives rise to the natural dynamic programming algorithm (e.g., see [26, Ch. 6]) for choosing $\lambda_0, \ldots, \lambda_{s-1}$ that runs $O(sb)$ time, treating all other parameters as constants.

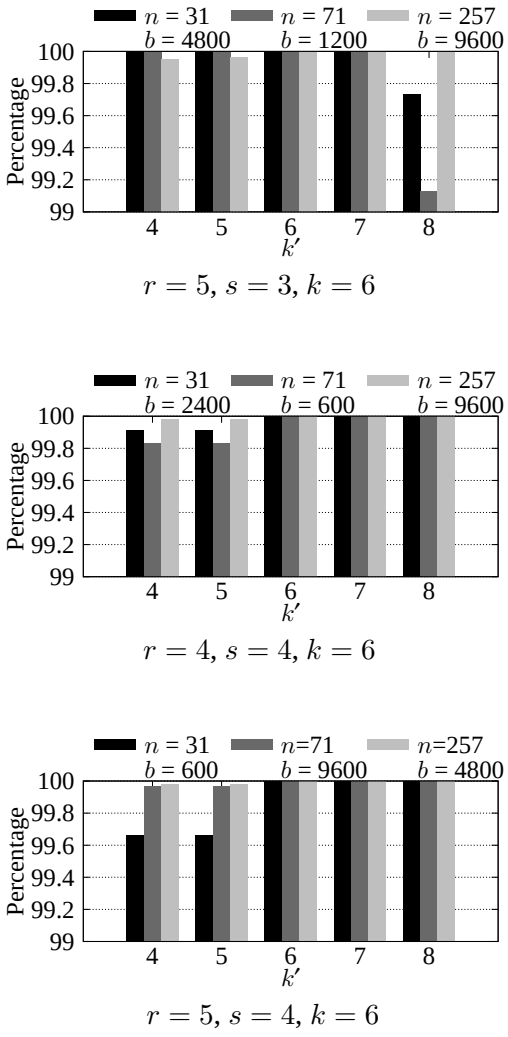


Figure 3.3: $\frac{lbAvail^{\mathsf{co}}(\lambda_0,\ldots,\lambda_{s-1})}{lbAvail^{\mathsf{co}}(\lambda'_0,\ldots,\lambda'_{s-1})}$ expressed as a percentage

3.2.2.2 Sensitivity to Choice of k

A potential disadvantage of the $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement strategy, or more precisely of the algorithm described in Section 3.2.2.1 to configure $\lambda_0, \ldots, \lambda_{s-1}$ for optimal availability, is that it does so only for the specified value k. A concern is that a $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement π configured for k node failures might fare poorly when subjected to $k' \neq k$ failures, at least in comparison to its availability were it configured for k' failures. This could occur if the $\lambda_0, \ldots, \lambda_{s-1}$ resulting from the configuration with k and those values resulting from configuration with k' were different.

	r			
n	2	3	4	5
31	$n_1 = 31$	$n_1 = 31$ [58] $n_2 = 31$	$n_1 = 28$ [29] $n_2 = 28$ [44] $n_3 = 31$	$n_1 = 25$ [29] $n_2 = 26$ [43] $n_3 = 23$ [69] $n_4 = 31$
71	$n_1 = 71$	$n_1 = 69$ [58] $n_2 = 71$	$n_1 = 70$ [29] $n_2 = 70$ [44] $n_3 = 71$	$n_1 = 65$ [29] $n_2 = 65$ [21] $n_3 = 71$ [69] $n_4 = 71$
257	$n_1 = 257$	$n_1 = 255$ [58] $n_2 = 257$	$n_1 = 256$ [29] $n_2 = 256$ [44] $n_3 = 257$	$n_1 = 245$ [29] $n_2 = 257$ [67] $n_3 = 243$ [69] $n_4 = 257$

Figure 3.4: Values of n_x used in this chapter

We have explored parameter spaces of interest to identify settings for which $\lambda_0, \ldots, \lambda_{s-1}$ would be different when configured for k or k' failed nodes, and then compared the resulting availability lower bounds. Figure 3.3 shows some representative examples. This figure plots the ratio $\frac{\mathit{lbAvail}^{\mathsf{co}}(\lambda_0,\ldots,\lambda_{s-1})}{\mathit{lbAvail}^{\mathsf{co}}(\lambda'_0,\ldots,\lambda'_{s-1})}$ expressed as a percentage for a $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement configured for k node failures and a $\mathsf{ComboOverlap}(\lambda'_0, \ldots, \lambda'_{s-1})$ placement configured for k' node failures. As such, when $k' = k$ this ratio will be 100%, for example. As this plot indicates, for some parameter values, this ratio dips below 100%, though we have not found cases in parameter regions of interest where this ratio drops below 98%.

3.2.3 Parameter Selection

Creating a $\mathsf{SimpleOverlap}(x, \lambda)$ placement for a set of n nodes can be achieved by identifying an $n_x \leq n$ and a μ_x that divides λ, for which $\mu_x\binom{n_x}{x+1}/\binom{r}{x+1}$ is integral and, moreover, a $\mathsf{SimpleOverlap}(x, \mu_x)$ placement exists for any $b \leq \mu_x\binom{n_x}{x+1}/\binom{r}{x+1}$ objects (see Observation 1). For such an n_x and μ_x, a $\mathsf{SimpleOverlap}(x, \mu_x)$ placement corresponds to a $(x+1)$-(n_x, r, μ_x)-*design* [61]. The study of the existence of such constructs is a fundamental question in design theory (e.g., [58]).

The need for μ_x to divide λ can be discharged if $\mu_x = 1$, in which case a $(x+1)$-(n_x, r, μ_x)-design is a Steiner system. Letting q be any prime power and for any $d \geq 2$, known infinite designs include [21]: $x + 1 = 2$, $r = q$, and $n_x = q^d$; $x + 1 = 3$, $r = q + 1$, and $n_x = q^d + 1$; $x + 1 = 2$, $r = q + 1$, and $n_x = q^d + \cdots + q + 1$; $x + 1 = 2$, $r = q + 1$, and $n_x = q^3 + 1$; and $x + 1 = 2$,

$r = 2^d$, and $n_x = 2^{d+d'} + 2^d - 2^{d'}$ for any $d' > d$. In addition, there are numerous known finite designs for $x < 5$, as surveyed by Colbourn and Mathon [21]. Known designs of Steiner systems suffice to implement $\mathsf{SimpleOverlap}(x, \lambda)$ for a wide array of practical parameter values, including all of the parameter settings investigated in this chapter. Figure 3.4 shows the Steiner systems used in our evaluations, as well as citations to where they can be found. (Note that when $x + 1 = r$, the constraints for a Steiner system are vacuously satisfied by sets of size r.)

As discussed in Observation 2, if a suitable $n_x \approx n$ cannot be found, then an alternative is to deconstruct the n nodes into "chunks" of size $n_{x1}, \ldots, n_{xm}$, each admitting a $\mathsf{SimpleOverlap}(x, \mu_{xi})$ placement, and to build a $\mathsf{SimpleOverlap}(x, \mu_x)$ placement for $\mu_x = \mathsf{lcm}\{\mu_{x1}, \ldots, \mu_{xm}\}$ on $\sum_{i=1}^{m} n_{xi}$ nodes by building a $\mathsf{SimpleOverlap}(x, \mu_x)$ placement on each chunk of n_{xi} nodes individually. This observation introduces a wide range of placement options for arbitrary n. This is demonstrated in Figure 3.5 for $\mu_x = 1$, which explores possible placements when even only $m = 3$. Each CDF shows the fraction of n values in the range $[50, 800]$ for which the "capacity gap" is at most the value on the horizontal axis, where the "capacity gap" is the difference between the ideal capacity (i.e., $\left\lfloor \mu_x \binom{n}{x+1} / \binom{r}{x+1} \right\rfloor$) and the capacity achievable (using concrete Steiner systems) by decomposing n into up to $m = 3$ chunks (i.e., $\sum_{i=1}^{m} \mu_{xi} \binom{n_{xi}}{x+1} / \binom{r}{x+1}$ with each $\mu_{xi} = 1$) expressed as a fraction of the ideal capacity. As shown there, in the cases $r \in \{2, 3, 4\}$, a very low (i.e., good) capacity gap can be achieved for nearly all system sizes n and all values of x. This is not the case for $r = 5$, however, where only about 10% of the system sizes n admit constructions (of which we are aware) for $x = 2$ or $x = 3$ with up to $m = 3$ chunks that yield a reasonably small capacity gap.

One way to address difficult cases like these (i.e., $r = 5$ along with $x = 2$ or $x = 3$) is to simply select one's system size n from the fraction of possible system sizes for which a small capacity gap can be achieved. Another alternative, however, is to expand consideration to $\mu_x > 1$, in which case numerous additional constructions are possible. Trivially, for any $x + 1 \leq r$, the collection of all r-subsets of n_x nodes suffices as a $\mathsf{SimpleOverlap}(x, \mu_x)$ placement for $\mu_x = \binom{n_x - x - 1}{r - x - 1}$. There are many other classes of $(x + 1)$-(n_x, r, μ_x)-designs with $\mu_x > 1$, as have been surveyed elsewhere [60, 1, 53]. In particular, Khosrovshahi and Laue [53, Table 4.3.7] survey a number of infinite designs for $3 \leq x + 1 \leq 5$.

To see the power of permitting $\mu_x > 1$ for realistic parameter settings, in Figure 3.6 we re-plot the $x = 2$ and $x = 3$ cases for $r = 5$ but allowing μ_x to be $\mu_x \leq 5$ (left) or $\mu_x \leq 10$ (right). As

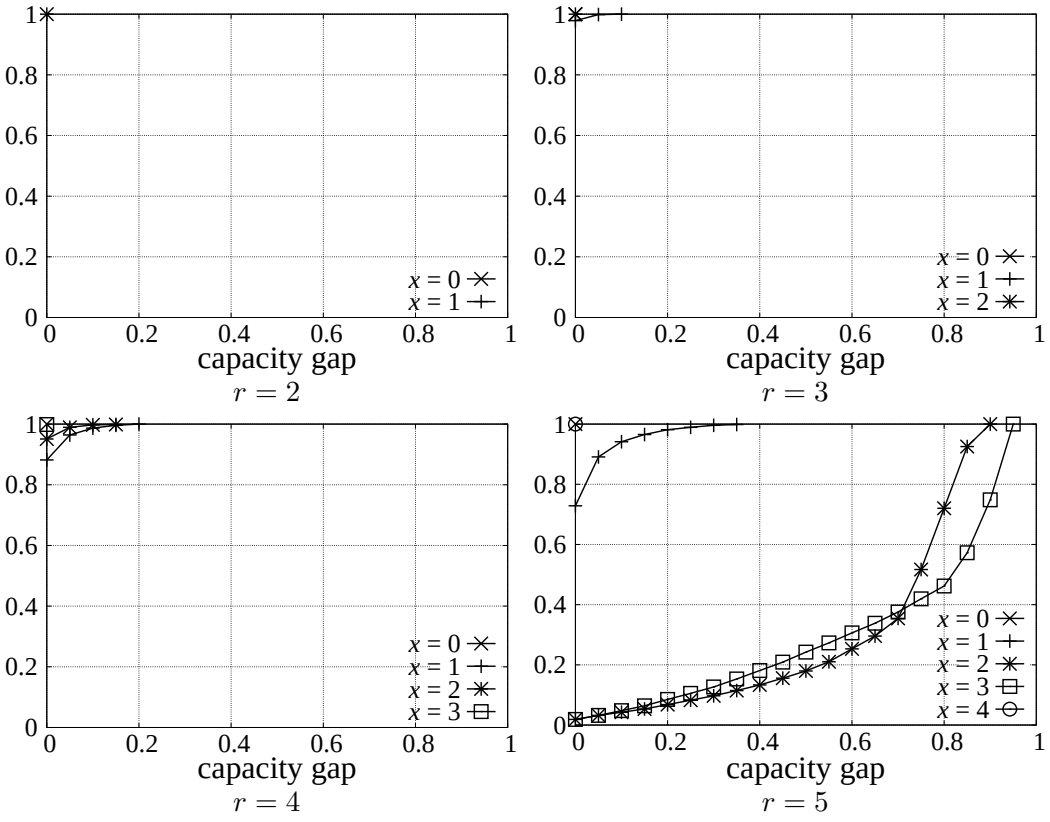


Figure 3.5: CDFs showing the fraction of system sizes $n \in [50, 800]$ for which the capacity gap (indicated on the horizontal axis, where lower is better) can be achieved using up to $m = 3$ Steiner systems ($\mu_{xi} = 1$)

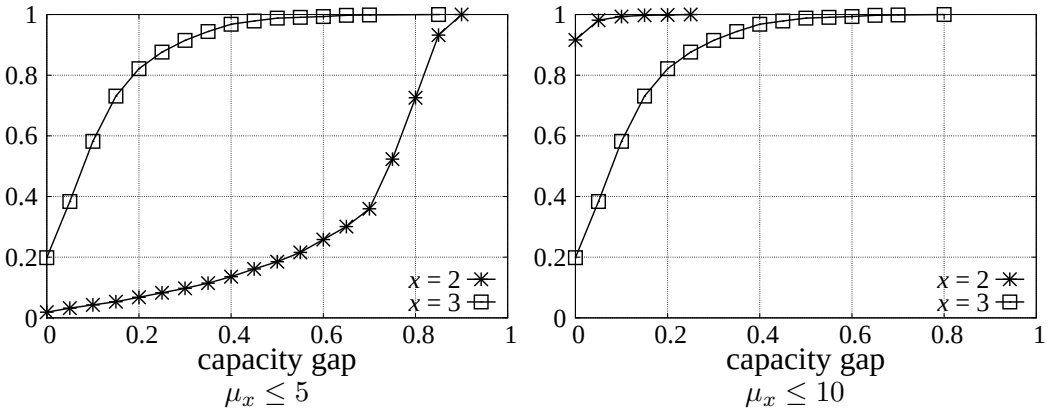


Figure 3.6: Re-plot of Figure 3.5 for $r = 5$ and $x \in \{2, 3\}$, but allowing $\mu_x = \mathsf{lcm}\{\mu_{x1}, \ldots, \mu_{xm}\} \geq 1$

can be seen in Figure 3.6, allowing $\mu_x \leq 5$ yields significant improvements in the $x = 3$ case, and permitting $\mu_x \leq 10$ additionally improves the $x = 2$ case dramatically. As such, permitting even modest growth of μ_x can greatly shrink the capacity gap in difficult cases.

3.3 Comparison to Random Replica Placement

As discussed in Section 3.1, the work of Yu and Gibbons [91] highlighted random replica placements as being very effective in ensuring the completion of multi-object operations when some objects' loss could be tolerated, in a system model where nodes fail independently with fixed probability. Given this result and the more general prominence of random replica placement in the research literature, we compare the availability offered by our ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement strategy to that offered by random replica placement. Specifically, we compare to a random placement strategy that (as in Yu and Gibbon's work) is load-balanced, where the average number of replicas per node is $\ell = \frac{rb}{n}$.

Definition 4. The Random placement strategy locates object replicas using a placement chosen uniformly at random from all placements that locate at most $\lceil \ell \rceil$ replicas on each node.

3.3.1 The Worst-Case Availability of Random

Evaluating the worst-case availability of Random placement is more subtle than for our previous placements, since *any* (load-balanced) placement can result from this placement strategy. So, in the truly worst case, Random would produce the worst possible placement for availability. That said, Random would do so with very low probability, and so this does not provide a representative view of how Random fares.

A more representative evaluation would take into account the *expected* behavior of the placement strategy. In some sense, the previous work of Yu and Gibbons [91] did so, but they did not take into account the *worst-case* behavior of the *adversary*. That is, their adversary failed nodes independently with a fixed probability, but ours adaptively chooses which nodes to fail based on the placement. So, to quantify the availability offered by Random in this worst case, we start by defining the *vulnerability* of Random:[1]

Definition 5. For any f, the vulnerability of Random, denoted $\mathit{Vuln}^{\mathsf{rnd}}(f)$, is the expected number of pairs $(\mathcal{K}, \mathcal{F})$ where $\mathcal{K} \subseteq \mathcal{N}$, $|\mathcal{K}| = k$, $\mathcal{F} \subseteq \mathcal{O}$, $|\mathcal{F}| \geq f$, and at least s replicas of each object in $\mathcal{F}$ are placed on the nodes in $\mathcal{K}$. The expectation is taken with respect to the random choices made by the Random placement strategy.

[1] Definition 5 and Definition 6 trivially generalize to any randomized placement strategy.

If $\mathit{Vuln}^{\mathsf{rnd}}(f) \geq 1$, then in expectation, there will be a set of k nodes that, if failed, will fail a set of at least f objects. It is then natural to define the number of objects that are *probably available* as follows:

Definition 6. In a Random placement, the number of objects that are probably available is

$$\mathit{prAvail}^{\mathsf{rnd}} = b - \max\{f : \mathit{Vuln}^{\mathsf{rnd}}(f) \geq 1\}$$

We now seek to quantify the probable availability of Random.

Theorem 7. As $\ell \longrightarrow \infty$,

$$\mathit{Vuln}^{\mathsf{rnd}}(f) \longrightarrow \binom{n}{k}\binom{n}{r}^{-b}\left(\sum_{f'=f}^{b}\binom{b}{f'}\alpha(n,k,r,s)^{f'}\left(\binom{n}{r}-\alpha(n,k,r,s)\right)^{b-f'}\right)$$

where $\alpha(n,k,r,s) = \sum_{s'=s}^{\min\{r,k\}}\binom{k}{s'}\binom{n-k}{r-s'}$.

Proof. Consider a variant Random$'$ of the Random placement in which the r replicas of each object are placed on r distinct nodes selected uniformly at random, but without limiting the number of replicas placed at each node. Let $X_{\mathsf{nd},\mathsf{obj}}$ be an indicator random variable defined as $X_{\mathsf{nd},\mathsf{obj}} = 1$ if a replica of obj is placed at nd and $X_{\mathsf{nd},\mathsf{obj}} = 0$ otherwise. Let $L_{\mathsf{nd}} = \sum_{\mathsf{obj}\in\mathcal{O}} X_{\mathsf{nd},\mathsf{obj}}$; i.e., L_{nd} is a random variable denoting the number of replicas placed at node nd.

While Random enforces that the number of replicas placed on each node is at most $\lceil\ell\rceil$, Random$'$ allows more. Specifically, for a fixed nd, $\{X_{\mathsf{nd},\mathsf{obj}}\}_{\mathsf{obj}\in\mathcal{O}}$ are independent, identically distributed Bernoulli random variables; i.e., $X_{\mathsf{nd},\mathsf{obj}} \sim B(\frac{r}{n})$ for each obj $\in \mathcal{O}$. Therefore, $\mathbb{E}(L_{\mathsf{nd}}) = \frac{br}{n} = \ell$ and, applying well-known Chernoff bounds (see, e.g., [62, Corollary 4.6]),

$$\mathbb{P}(|L_{\mathsf{nd}} - \ell| \geq \delta\ell) \leq 2e^{-\ell\delta^2/3}$$

for any $0 < \delta < 1$. Consequently, the distribution of object replicas to nodes under Random$'$ (quickly) approaches the distribution induced by Random as $\ell \longrightarrow \infty$, and so we can reason about the asymptotic distribution induced by Random using the one induced by Random$'$.

Let $\mathbf{failedNodes}(\mathcal{K})$ denote the event that set $\mathcal{K} \subseteq \mathcal{N}$ is the complete set of failed nodes, and $\mathbf{failedObjs}(\mathcal{F})$ denote the event that the set $\mathcal{F} \subseteq \mathcal{O}$ is the complete set of objects that failed due to the failure of the nodes in $\mathcal{K}$. Now, under Random',

$$
\begin{aligned}
&\mathbb{P}\big(\mathbf{failedObjs}(\mathcal{F}) \;\big|\; \mathbf{failedNodes}(\mathcal{K})\big) \\
&= \left[\prod_{\mathsf{obj}\in\mathcal{F}} \mathbb{P}\binom{\text{obj replicas placed on } s' \geq s}{\text{nodes in } \mathcal{K} \text{ and } r-s' \text{ others}}\right] \cdot \left[\prod_{\mathsf{obj}\in\mathcal{O}\setminus\mathcal{F}} \mathbb{P}\binom{\text{obj replicas placed on } s' < s}{\text{nodes in } \mathcal{K} \text{ and } r-s' \text{ others}}\right] \\
&= \left[\prod_{\mathsf{obj}\in\mathcal{F}} \sum_{s'=s}^{\min\{r,k\}} \frac{\binom{k}{s'}\binom{n-k}{r-s'}}{\binom{n}{r}}\right] \cdot \left[\prod_{\mathsf{obj}\in\mathcal{O}\setminus\mathcal{F}} \sum_{s'=0}^{s-1} \frac{\binom{k}{s'}\binom{n-k}{r-s'}}{\binom{n}{r}}\right] \\
&= \binom{n}{r}^{-b} \left(\sum_{s'=s}^{\min\{r,k\}} \binom{k}{s'}\binom{n-k}{r-s'}\right)^{f} \left(\sum_{s'=0}^{s-1} \binom{k}{s'}\binom{n-k}{r-s'}\right)^{b-f} \\
&= \binom{n}{r}^{-b} \alpha(n,k,r,s)^{f} \left(\binom{n}{r} - \alpha(n,k,r,s)\right)^{b-f} \qquad (3.9)
\end{aligned}
$$

To complete the proof, for any $\mathcal{K} \subseteq \mathcal{N}$, $|\mathcal{K}| = k$, and any $\mathcal{F} \subseteq \mathcal{O}$, define an indicator random variable $X_{\mathcal{K},\mathcal{F}}$ as follows: $X_{\mathcal{K},\mathcal{F}} = 1$ if $\mathcal{F}$ is the set of objects failed when the nodes $\mathcal{K}$ fail, and $X_{\mathcal{K},\mathcal{F}} = 0$ otherwise. The expected value of $X_{\mathcal{K},\mathcal{F}}$ is then

$$\mathbb{E}\left(X_{\mathcal{K},\mathcal{F}}\right) = \mathbb{P}\big(\mathbf{failedObjs}(\mathcal{F}) \;\big|\; \mathbf{failedNodes}(\mathcal{K})\big)$$

By linearity of expectation, $\mathit{Vuln}^{\mathsf{rnd}}(f)$ is then:

$$\mathit{Vuln}^{\mathsf{rnd}}(f) = \mathbb{E}\left(\sum_{\substack{\mathcal{K}\subseteq\mathcal{N}:\\ |\mathcal{K}|=k}} \sum_{\substack{\mathcal{F}\subseteq\mathcal{O}:\\ |\mathcal{F}|\geq f}} X_{\mathcal{K},\mathcal{F}}\right) = \sum_{\substack{\mathcal{K}\subseteq\mathcal{N}:\\ |\mathcal{K}|=k}} \sum_{\substack{\mathcal{F}\subseteq\mathcal{O}:\\ |\mathcal{F}|\geq f}} \mathbb{E}\left(X_{\mathcal{K},\mathcal{F}}\right)$$

Plugging in Equation 3.9 yields the result. □

For the rest of this chapter, we use the limit of $\mathit{Vuln}^{\mathsf{rnd}}(f)$ given in Theorem 7 to calculate $\mathit{prAvail}^{\mathsf{rnd}}$ as defined in Definition 6. By comparing $\mathit{prAvail}^{\mathsf{rnd}}$ to simulation results for parameter ranges of interest that are feasible to simulate, we found that once $b \geq 600$, $\mathit{prAvail}^{\mathsf{rnd}}$ has converged to within 10% of the empirical average of $\mathit{Avail}(\pi)$ for Random placements π. So, in

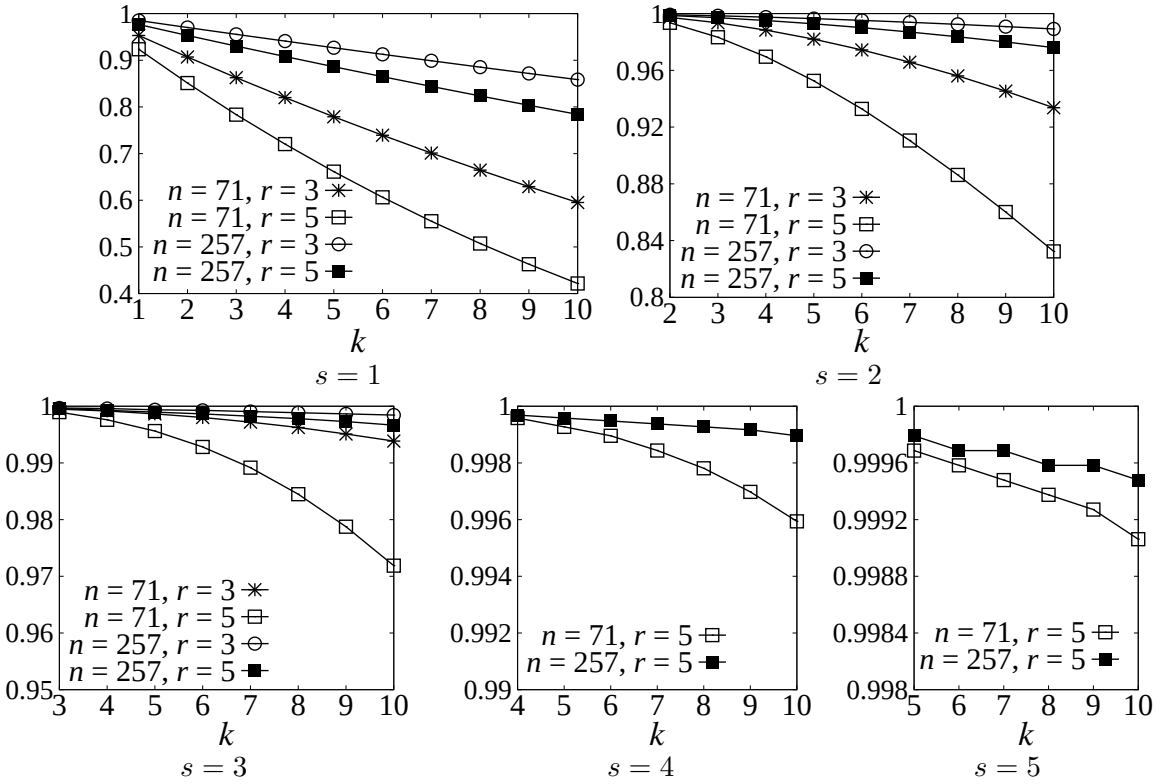


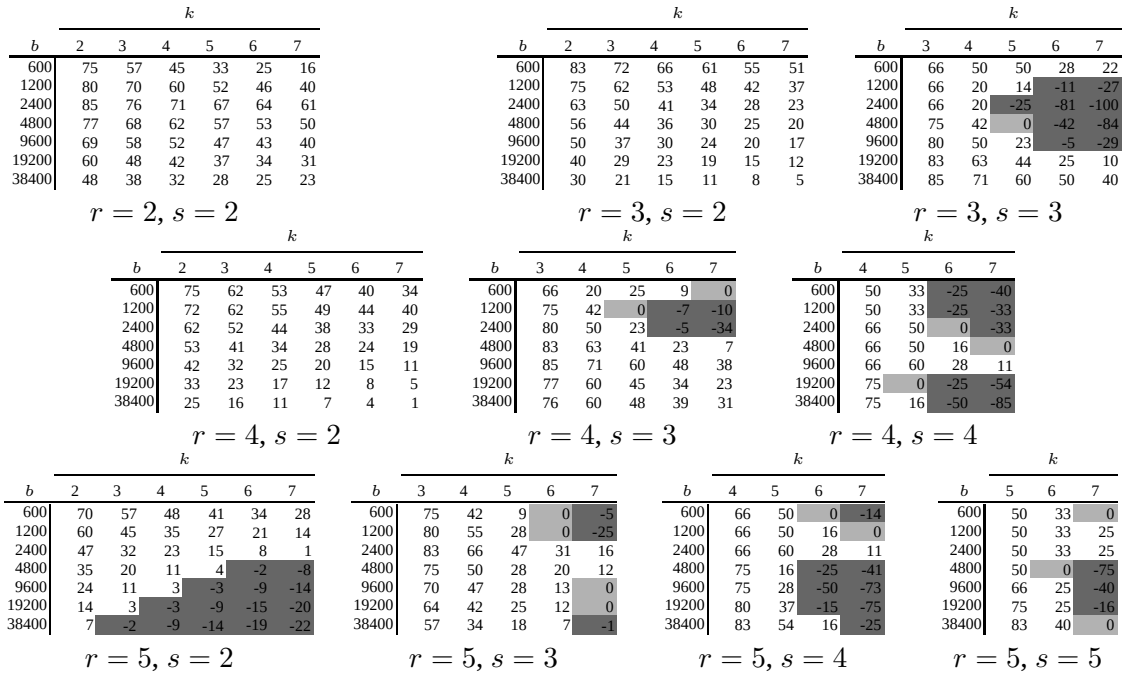
Figure 3.7: $\frac{1}{b}$*prAvail* $^{\mathsf{rnd}}$ for $b = 38400$

drawing our comparisons between Random and ComboOverlap placements, we will restrict our attention to $b \geq 600$, to be fair to Random.

Figure 3.7 plots $\frac{1}{b}$*prAvail* $^{\mathsf{rnd}}$, i.e., *prAvail* $^{\mathsf{rnd}}$ as a fraction of b, for various values of s, r, and n when $b = 38400$. Plotted in this way as a fraction of b, the curves look very similar for the various values of b that we have explored. One takeaway from these graphs is that the case $s = 1$ performs quite poorly relative to larger s (notice the vertical axes are not the same scale), and we prove in Section 3.3.4 that this is true for Random placements in general.

3.3.2 Comparison Results

We now compare ComboOverlap and Random placements using *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) -$ *prAvail* $^{\mathsf{rnd}}$ across a range of parameter settings, i.e., using a ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement computed to maximize *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ (Section 3.2.2.1). We use the limit of *Vuln*$^{\mathsf{rnd}}(f)$ in Theorem 7 to get *prAvail* $^{\mathsf{rnd}}$ as defined in Definition 6. The measure *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) -$ *prAvail* $^{\mathsf{rnd}}$ is conservative in the sense that *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ is a *lower bound*, whereas



$r = 2, s = 2$

b	k 2	3	4	5	6	7
600	75	57	45	33	25	16
1200	80	70	60	52	46	40
2400	85	76	71	67	64	61
4800	77	68	62	57	53	50
9600	69	58	52	47	43	40
19200	60	48	42	37	34	31
38400	48	38	32	28	25	23

$r = 3, s = 2$

b	k 2	3	4	5	6	7
600	83	72	66	61	55	51
1200	75	62	53	48	42	37
2400	63	50	41	34	28	23
4800	56	44	36	30	25	20
9600	50	37	30	24	20	17
19200	40	29	23	19	15	12
38400	30	21	15	11	8	5

$r = 3, s = 3$

b	k 3	4	5	6	7
600	66	50	50	28	22
1200	66	20	14	-11	-27
2400	66	20	-25	-81	-100
4800	75	42	0	-42	-84
9600	80	50	23	-5	-29
19200	83	63	44	25	10
38400	85	71	60	50	40

$r = 4, s = 2$

b	k 2	3	4	5	6	7
600	75	62	53	47	40	34
1200	72	62	55	49	44	40
2400	62	52	44	38	33	29
4800	53	41	34	28	24	19
9600	42	32	25	20	15	11
19200	33	23	17	12	8	5
38400	25	16	11	7	4	1

$r = 4, s = 3$

b	k 3	4	5	6	7
600	66	20	25	9	0
1200	75	42	0	-7	-10
2400	80	50	23	-5	-34
4800	83	63	41	23	7
9600	85	71	60	48	38
19200	77	60	45	34	23
38400	76	60	48	39	31

$r = 4, s = 4$

b	k 4	5	6	7
600	50	33	-25	-40
1200	50	33	-25	-33
2400	66	50	0	-33
4800	66	50	16	0
9600	66	60	28	11
19200	75	0	-25	-54
38400	75	16	-50	-85

$r = 5, s = 2$

b	k 2	3	4	5	6	7
600	70	57	48	41	34	28
1200	60	45	35	27	21	14
2400	47	32	23	15	8	1
4800	35	20	11	4	-2	-8
9600	24	11	3	-3	-9	-14
19200	14	3	-3	-9	-15	-20
38400	7	-2	-9	-14	-19	-22

$r = 5, s = 3$

b	k 3	4	5	6	7
600	75	42	9	0	-5
1200	80	55	28	0	-25
2400	83	66	47	31	16
4800	75	50	28	20	12
9600	70	47	28	13	0
19200	64	42	25	12	0
38400	57	34	18	7	-1

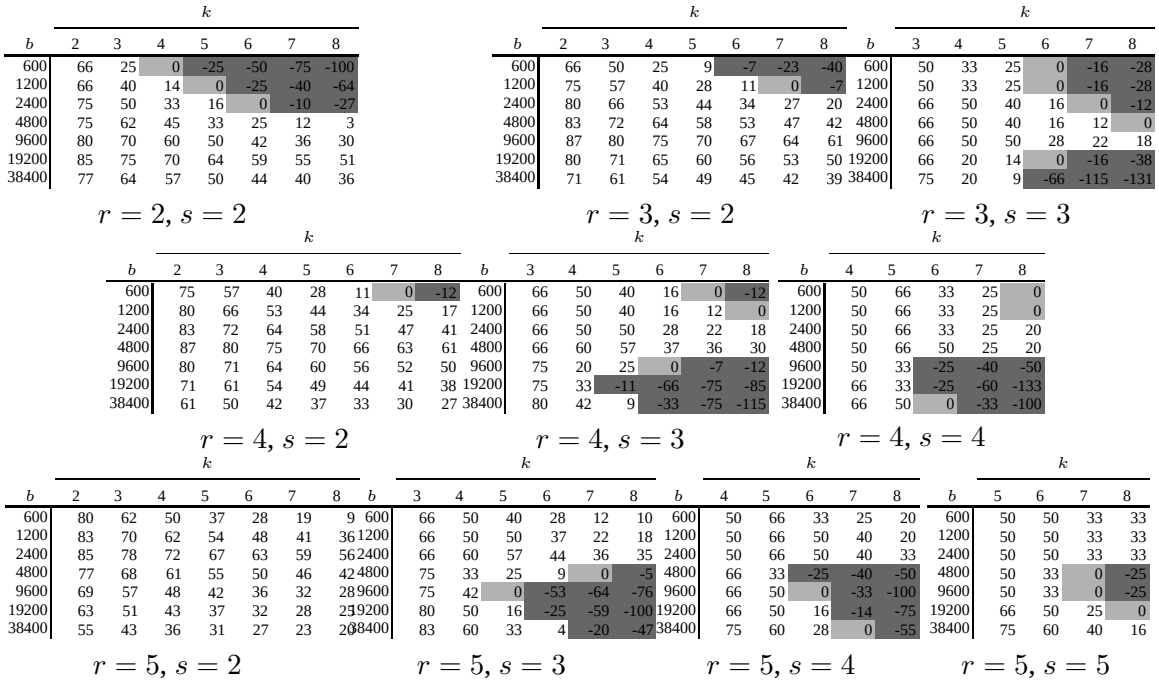
$r = 5, s = 4$

b	k 4	5	6	7
600	66	50	0	-14
1200	66	50	16	0
2400	66	60	28	11
4800	75	16	-25	-41
9600	75	28	-50	-73
19200	80	37	-15	-75
38400	83	54	16	-25

$r = 5, s = 5$

b	k 5	6	7
600	50	33	0
1200	50	33	25
2400	50	33	25
4800	50	0	-75
9600	66	25	-40
19200	75	25	-16
38400	83	40	0

(a) $n = 71$



$r = 2, s = 2$

b	k 2	3	4	5	6	7	8
600	66	25	0	-25	-50	-75	-100
1200	66	40	14	0	-25	-40	-64
2400	75	50	33	16	0	-10	-27
4800	75	62	45	33	25	12	3
9600	80	70	60	50	42	36	30
19200	85	75	70	64	59	55	51
38400	77	64	57	50	44	40	36

$r = 3, s = 2$

b	k 2	3	4	5	6	7	8
600	66	50	25	9	-7	-23	-40
1200	75	57	40	28	11	0	-7
2400	80	66	53	44	34	27	20
4800	83	72	64	58	53	47	42
9600	87	80	75	70	67	64	61
19200	80	71	65	60	56	53	50
38400	71	61	54	49	45	42	39

$r = 3, s = 3$

b	k 3	4	5	6	7	8
600	50	33	25	0	-16	-28
1200	50	33	25	0	-16	-28
2400	66	50	40	16	0	-12
4800	66	50	40	16	12	0
9600	66	50	50	28	22	18
19200	66	20	14	0	-16	-38
38400	75	20	9	-66	-115	-131

$r = 4, s = 2$

b	k 2	3	4	5	6	7	8
600	75	57	40	28	11	0	-12
1200	80	66	53	44	34	25	17
2400	83	72	64	58	51	47	41
4800	87	80	75	70	66	63	61
9600	80	71	64	60	56	52	50
19200	71	61	54	49	44	41	38
38400	61	50	42	37	33	30	27

$r = 4, s = 3$

b	k 3	4	5	6	7	8
600	66	50	40	16	0	-12
1200	66	50	40	16	12	0
2400	66	50	50	28	22	18
4800	66	60	57	37	36	30
9600	75	20	25	0	-7	-12
19200	75	33	-11	-66	-75	-85
38400	80	42	9	-33	-75	-115

$r = 4, s = 4$

b	k 4	5	6	7	8
600	50	66	33	25	0
1200	50	66	33	25	0
2400	50	66	33	25	20
4800	50	66	50	25	20
9600	50	33	-25	-40	-50
19200	66	33	-25	-60	-133
38400	66	50	0	-33	-100

$r = 5, s = 2$

b	k 2	3	4	5	6	7	8
600	80	62	50	37	28	19	9
1200	83	70	62	54	48	41	36
2400	85	78	72	67	63	59	56
4800	77	68	61	55	50	46	42
9600	69	57	48	42	36	32	28
19200	63	51	43	37	32	28	25
38400	55	43	36	31	27	23	20

$r = 5, s = 3$

b	k 3	4	5	6	7	8
600	66	50	40	28	12	10
1200	66	50	50	37	22	18
2400	66	60	57	44	36	35
4800	75	33	25	9	0	-5
9600	75	42	0	-53	-64	-76
19200	80	50	16	-25	-59	-100
38400	83	60	33	4	-20	-47

$r = 5, s = 4$

b	k 4	5	6	7	8
600	50	66	33	25	20
1200	50	66	50	40	20
2400	50	66	50	40	33
4800	66	33	-25	-40	-50
9600	66	50	0	-33	-100
19200	66	50	16	-14	-75
38400	75	60	28	0	-55

$r = 5, s = 5$

b	k 5	6	7	8
600	50	50	33	33
1200	50	50	33	33
2400	50	50	33	33
4800	50	33	0	-25
9600	50	33	0	-25
19200	66	50	25	0
38400	75	60	40	16

(b) $n = 257$

Figure 3.8: $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) - \mathit{prAvail}^{\mathsf{rnd}}$ for an optimal $\mathsf{ComboOverlap}(\lambda_0, \ldots, \lambda_{s-1})$ placement as a percentage of the maximum possible improvement $b - \mathit{prAvail}^{\mathsf{rnd}}$

$\mathit{prAvail}^{\mathsf{rnd}}$ is only a probabilistic estimate of the number of objects that remain available under Random and so it is not guaranteed.

Here and elsewhere in this chapter, we use $n \in \{31, 71, 257\}$, both because these values span a reasonably wide range and because suitable $n_x \approx n$ and μ_x can be found for them without resorting to Observation 2. (These are by no means the only values that meet these criteria, though.) In particular, this means that the ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placements represented in this section have concrete implementations. The selection of each n_x (with $\mu_x = 1$) is detailed in Section 3.2.3, as is an exploration of Observation 2.

A summary of results is given in Figure 3.8, where the top (Figure 3.8a) shows the results with $n = 71$ and the bottom (Figure 3.8b) shows the results with $n = 257$. Each portion shows a table for $2 \leq r \leq 5$ and $2 \leq s \leq r$. (The case $s = 1$ is further discussed in Section 3.3.4.) The number k of failed nodes is ranged over $s \leq k \leq 7$ in the $n = 71$ case, and over $s \leq k \leq 8$ in the $n = 257$ case; both ranges encompass a substantial rate of node failures. In each table, the number b of objects begins at $b = 600$ and is repeatedly doubled until it reaches $b = 38400$. Each table entry indicates $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) - \mathit{prAvail}^{\mathsf{rnd}}$ as a percentage of the maximum possible improvement $b - \mathit{prAvail}^{\mathsf{rnd}}$ that could be achieved over $\mathit{prAvail}^{\mathsf{rnd}}$. To ease readability, cells where $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) > \mathit{prAvail}^{\mathsf{rnd}}$ (and so ComboOverlap "wins") are colored white; cells where $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) = \mathit{prAvail}^{\mathsf{rnd}}$ (neither ComboOverlap nor Random "wins") are colored light gray; and cells where $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) < \mathit{prAvail}^{\mathsf{rnd}}$ (Random "wins") are colored dark gray.

It is evident upon a cursory glance that ComboOverlap "wins" most of the time, and the percentage by which it does so is often very substantial. For example, the table in the very upper-left corner of Figure 3.8a indicates that in the case $n = 71$, $r = 2$, $s = 2$, $b = 2400$ and $k = 2$, ComboOverlap *guarantees* to preserve the availability of 85% of the objects that will fail in expectation under Random.

Since each ComboOverlap placement is a combination of SimpleOverlap(x, λ_x) placements, in Section 3.3.3 we show the contribution of each SimpleOverlap(x, λ_x) placement to the final ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement for $n = 71$ (Figure 3.10) and $n = 257$ (Figure 3.11), as well as for $n = 31$ (Figure 3.9). (The $n = 31$ case is excluded from discussion in this section due to space limitations.) These figures do not show SimpleOverlap$(0, \lambda_0)$ placements to save space, since they contribute so minimally. In particular, the figures exclude breakdowns when $s = 2$, since in this case, only $x = 1$ contributes to ComboOverlap; i.e., SimpleOverlap$(1, \lambda_1)$ and ComboOverlap(λ_1)

are identically the same. Very briefly, we distill out the following observations from the figures in Section 3.3.3.

- When b grows and n and x are held constant, SimpleOverlap(x, λ) availability improves relative to Random until λ has to grow to satisfy Equation 3.1. This can be seen, for example, in the $r = s = 3$, $x = 1$ table in the upper-left corner of Figure 3.11 ($n = 257$). As shown there, while λ can remain at 1 (see rightmost column of leftmost table), SimpleOverlap(x, λ) (and so ComboOverlap, as shown in the rightmost table on the same row) "wins" more, but its performance diminishes as λ grows.
- One way to offset the need to grow λ is to adjust x, since when $k \approx s$, doing so impacts availability only a small amount (Equation 3.2) but can allow a SimpleOverlap(x, λ) placement to accommodate many more objects (assuming $n \gg r$). This is shown clearly in, e.g., the $r = s = 3$ cases in Figure 3.9 ($n = 31$) and Figure 3.10 ($n = 71$), where moving from $x = 1$ to $x = 2$ relieves the pressure on λ to increase, allowing the advantages of ComboOverlap to be preserved as b grows.
- Another way to slow the growth of λ is to increase n. For a fixed number b of objects and as n grows, ComboOverlap will increasingly select to place objects using SimpleOverlap(x, λ_x) placements for smaller x. To see this, compare the contributions of, e.g., $x = 1$ and $x = 2$ to the resulting ComboOverlap placement for $r = 3$, $s = 3$ in the top rows of Figure 3.9 ($n = 31$) and Figure 3.11 ($n = 257$). This can be explained by observing that as n and so each n_x grows, the smallest x that suffices to achieve Equation 3.1 can shrink while keeping λ the same. This, in turn, yields better availability (Equation 3.2).
- Even at specific parameter values, ComboOverlap can outperform SimpleOverlap(x, λ) for any single x. This is illustrated in the top row ($r = 3$, $s = 3$) of Figure 3.9, for example, in which at $b = 4800$ and $k \in \{5, 6\}$, the ComboOverlap table includes entries (44 and 36) that exceed the corresponding entries of any of the SimpleOverlap(x, λ_x) tables in its row. This occurs at a value of b at which SimpleOverlap$(2, \lambda_2)$ must increase λ_2 from $\lambda_2 = 1$ to $\lambda_2 = 2$ to satisfy Equation 3.1. In this case, it turns out to be better to build ComboOverlap using a SimpleOverlap$(2, 1)$ placement in conjunction with a SimpleOverlap$(1, 2)$ placement to satisfy Equation 3.4, rather than using a SimpleOverlap$(2, 2)$ placement alone. This advantage of ComboOverlap is not fre-

quently illustrated in Section 3.3.3, though testing more exhaustively with different values of b would elicit it more.

3.3.3 Breakdown of ComboOverlap Placements

Recall that ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement combines individual SimpleOverlap(x, λ_x) placements. In this section we detail for various parameters how individual SimpleOverlap(x, λ_x) placements contribute to the ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placements computed via the algorithm described in Section 3.2.2.1, or more specifically how they contribute to the results showing the improvement of ComboOverlap placements over Random placements in Section 3.3.2.

We demonstrate these contributions through Figures 3.9–3.11, which isolate three cases: $n = 31$ (Figure 3.9), $n = 71$ (Figure 3.10), and $n = 257$ (Figure 3.11). In each figure, each row corresponds to a particular setting for r and s. The rightmost table in each row represents the ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement for its row's r and s and, in the case $n = 71$ or $n = 257$, is an exact copy of the table in Figure 3.8a or Figure 3.8b, respectively, for the same r and s. (The $n = 31$ case was elided from Section 3.3.2 due to space limitations, though many of the ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ tables for $n = 31$ are included in Figure 3.9.) The other tables in its row represent SimpleOverlap(x, λ_x) placements for the same r and s. As in Figure 3.8, a white table cell indicates that for the parameter settings it represents, our placement outperforms (i.e., achieves better availability than) a Random placement; a light gray cell indicates that both perform equally well (setting aside the conservative nature of the comparison, see Section 3.3.2); and a dark gray cell indicates that Random provides (potentially) better availability.

3.3.4 The $s = 1$ Case

In our comparisons between ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ and Random placements in Section 3.3.2, we deferred the case $s = 1$. In this case, a ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement is just a SimpleOverlap$(0, \lambda_0)$ placement. Our analysis in this chapter applies to the $s = 1$ case, and a comparison using $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0) - \mathit{prAvail}^{\mathsf{rnd}}$ as in Section 3.3.2 indicates that Random slightly outperforms SimpleOverlap$(0, \lambda_0)$ in this measure, for the parameter values we tested. Nevertheless, we relegated this case to this section simply because both Random and SimpleOverlap$(0, \lambda_0)$ perform poorly in this case. The following lemma formalizes this claim for Random placements.

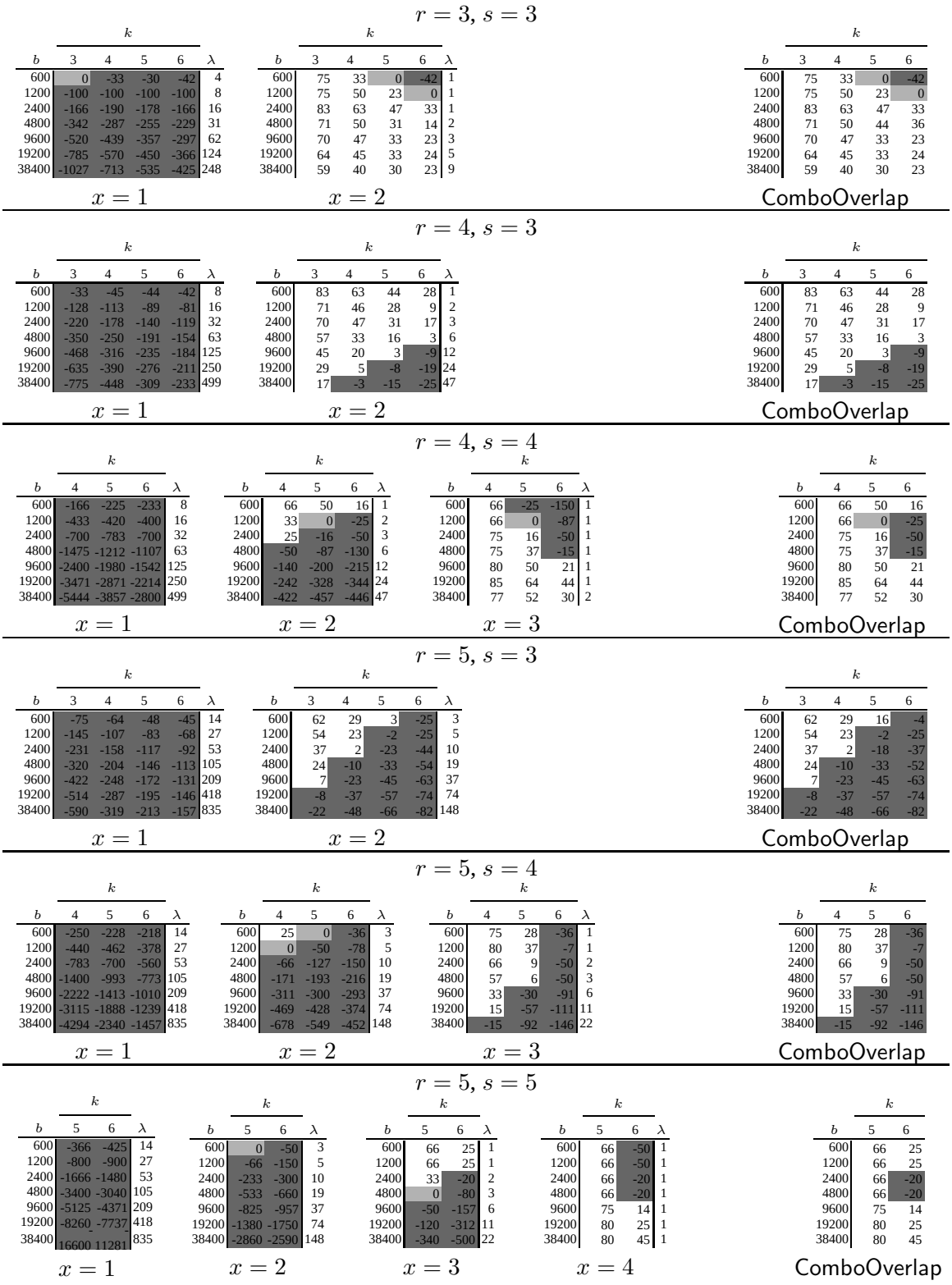


Figure 3.9: $\mathit{lbAvail}^{\mathsf{so}}(x, \lambda) - \mathit{prAvail}^{\mathsf{rnd}}$ for SimpleOverlap(x, λ) placements and $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) - \mathit{prAvail}^{\mathsf{rnd}}$ for best ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement (right most column) when $s > 2$, as a percentage of the maximum possible improvement $b - \mathit{prAvail}^{\mathsf{rnd}}$, when $n = 31$

$r = 3, s = 3$

b	3	4	5	6	7	λ
600	66	50	50	28	22	1
1200	33	20	14	-11	-27	2
2400	-33	-60	-62	-81	-100	4
4800	-75	-100	-130	-150	-157	7
9600	-160	-225	-230	-242	-237	13
19200	-316	-354	-361	-362	-348	25
38400	-614	-614	-564	-525	-493	50

$x = 1$

b	3	4	5	6	7	λ
600	66	0	-66	-185	-288	1
1200	66	20	-42	-122	-218	1
2400	66	20	-25	-81	-150	1
4800	75	42	0	-42	-84	1
9600	80	50	23	-5	-29	1
19200	83	63	44	25	10	1
38400	85	71	60	50	40	1

$x = 2$

b	3	4	5	6	7
600	66	50	50	28	22
1200	66	20	14	-11	-27
2400	66	20	-25	-81	-100
4800	75	42	0	-42	-84
9600	80	50	23	-5	-29
19200	83	63	44	25	10
38400	85	71	60	50	40

ComboOverlap

$r = 4, s = 3$

b	3	4	5	6	7	λ
600	33	20	25	9	0	2
1200	25	14	0	-7	-10	3
2400	-20	-50	-53	-57	-61	6
4800	-100	-118	-135	-130	-121	12
9600	-242	-242	-220	-207	-194	24
19200	-433	-380	-332	-293	-265	48
38400	-638	-540	-451	-384	-336	96

$x = 1$

b	3	4	5	6	7	λ
600	66	20	-25	-81	-150	1
1200	75	42	0	-42	-84	1
2400	80	50	23	-5	-34	1
4800	83	63	41	23	7	1
9600	85	71	60	48	38	1
19200	77	60	45	34	23	2
38400	76	60	48	39	31	3

$x = 2$

b	3	4	5	6	7
600	66	20	25	9	0
1200	75	42	0	-7	-10
2400	80	50	23	-5	-34
4800	83	63	41	23	7
9600	85	71	60	48	38
19200	77	60	45	34	23
38400	76	60	48	39	31

ComboOverlap

$r = 4, s = 4$

b	4	5	6	7	λ
600	0	0	-25	-40	2
1200	-50	-66	-75	-66	3
2400	-100	-150	-200	-250	6
4800	-300	-400	-400	-425	12
9600	-700	-700	-757	-833	24
19200	-1100	-1500	-1400	-1427	48
38400	-2300	-2566	-2300	-2300	96

$x = 1$

b	4	5	6	7	λ
600	50	33	-25	-60	1
1200	50	33	-25	-33	1
2400	66	50	0	-33	1
4800	66	50	16	0	1
9600	66	60	28	11	1
19200	50	0	-25	-54	2
38400	25	-16	-50	-85	3

$x = 2$

b	4	5	6	7	λ
600	50	-66	-275	-600	1
1200	50	-66	-275	-483	1
2400	66	-25	-200	-483	1
4800	66	-25	-150	-337	1
9600	66	0	-114	-288	1
19200	75	0	-87	-218	1
38400	75	16	-50	-150	1

$x = 3$

b	4	5	6	7
600	50	33	-25	-40
1200	50	33	-25	-33
2400	66	50	0	-33
4800	66	50	16	0
9600	66	60	28	11
19200	75	0	-25	-54
38400	75	16	-50	-85

ComboOverlap

$r = 5, s = 3$

b	3	4	5	6	7	λ
600	25	14	9	0	-5	3
1200	-20	-33	-42	-50	-50	6
2400	-100	-100	-110	-106	-100	12
4800	-200	-200	-185	-172	-162	24
9600	-370	-308	-271	-240	-216	47
19200	-564	-431	-362	-307	-269	93
38400	-780	-572	-454	-374	-317	185

$x = 1$

b	3	4	5	6	7	λ
600	75	42	9	-33	-75	1
1200	80	55	28	0	-25	1
2400	83	66	47	31	16	1
4800	75	50	28	20	12	2
9600	70	47	28	13	0	3
19200	64	42	25	12	0	5
38400	57	34	18	7	-1	9

$x = 2$

b	3	4	5	6	7
600	75	42	9	0	-5
1200	80	55	28	0	-25
2400	83	66	47	31	16
4800	75	50	28	20	12
9600	70	47	28	13	0
19200	64	42	25	12	0
38400	57	34	18	7	-1

ComboOverlap

$r = 5, s = 4$

b	4	5	6	7	λ
600	0	-25	-40	-42	3
1200	-100	-150	-150	-162	6
2400	-300	-300	-328	-366	12
4800	-500	-566	-650	-600	24
9600	-1075	-1014	-1070	-933	47
19200	-1760	-1837	-1684	-1525	93
38400	-2983	-2700	-2466	-2210	185

$x = 1$

b	4	5	6	7	λ
600	66	50	0	-14	1
1200	66	50	16	0	1
2400	66	60	28	11	1
4800	50	16	-25	-41	2
9600	25	0	-50	-73	3
19200	0	-50	-92	-115	5
38400	-50	-100	-150	-178	9

$x = 2$

b	4	5	6	7	λ
600	66	-25	-200	-400	1
1200	66	-25	-150	-337	1
2400	66	0	-114	-288	1
4800	75	16	-87	-191	1
9600	75	28	-50	-133	1
19200	80	37	-15	-75	1
38400	83	54	16	-25	1

$x = 3$

b	4	5	6	7
600	66	50	0	-14
1200	66	50	16	0
2400	66	60	28	11
4800	75	16	-25	-41
9600	75	28	-50	-73
19200	80	37	-15	-75
38400	83	54	16	-25

ComboOverlap

$r = 5, s = 5$

b	5	6	7	λ
600	-50	-33	-100	3
1200	-200	-200	-200	6
2400	-500	-500	-525	12
4800	-1100	-1100	-1150	24
9600	-1466	-1650	-1860	47
19200	-3000	-3375	-3150	93
38400	-6066	-6825	-5442	185

$x = 1$

b	5	6	7	λ
600	50	33	0	1
1200	50	33	25	1
2400	50	33	25	1
4800	0	-33	-75	2
9600	0	-50	-100	3
19200	-66	-150	-183	5
38400	-200	-350	-342	9

$x = 2$

b	5	6	7	λ
600	50	0	-133	1
1200	50	0	-133	1
2400	50	0	-75	1
4800	50	0	-75	1
9600	66	25	-40	1
19200	75	25	-16	1
38400	83	40	0	1

$x = 3$

b	5	6	7	λ
600	50	-100	-600	1
1200	50	-100	-425	1
2400	50	-100	-425	1
4800	50	-100	-425	1
9600	66	-50	-320	1
19200	75	-50	-250	1
38400	83	-20	-200	1

$x = 4$

b	5	6	7
600	50	33	0
1200	50	33	25
2400	50	33	25
4800	50	0	-75
9600	66	25	-40
19200	75	25	-16
38400	83	40	0

ComboOverlap

Figure 3.10: *lbAvail* $^{\mathsf{so}}(x, \lambda)$ − *prAvail* $^{\mathsf{rnd}}$ for SimpleOverlap(x, λ) placements and *lbAvail* $^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1})$ − *prAvail* $^{\mathsf{rnd}}$ for best ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement (right most column) when $s > 2$, as a percentage of the maximum possible improvement b − *prAvail* $^{\mathsf{rnd}}$, when $n = 71$

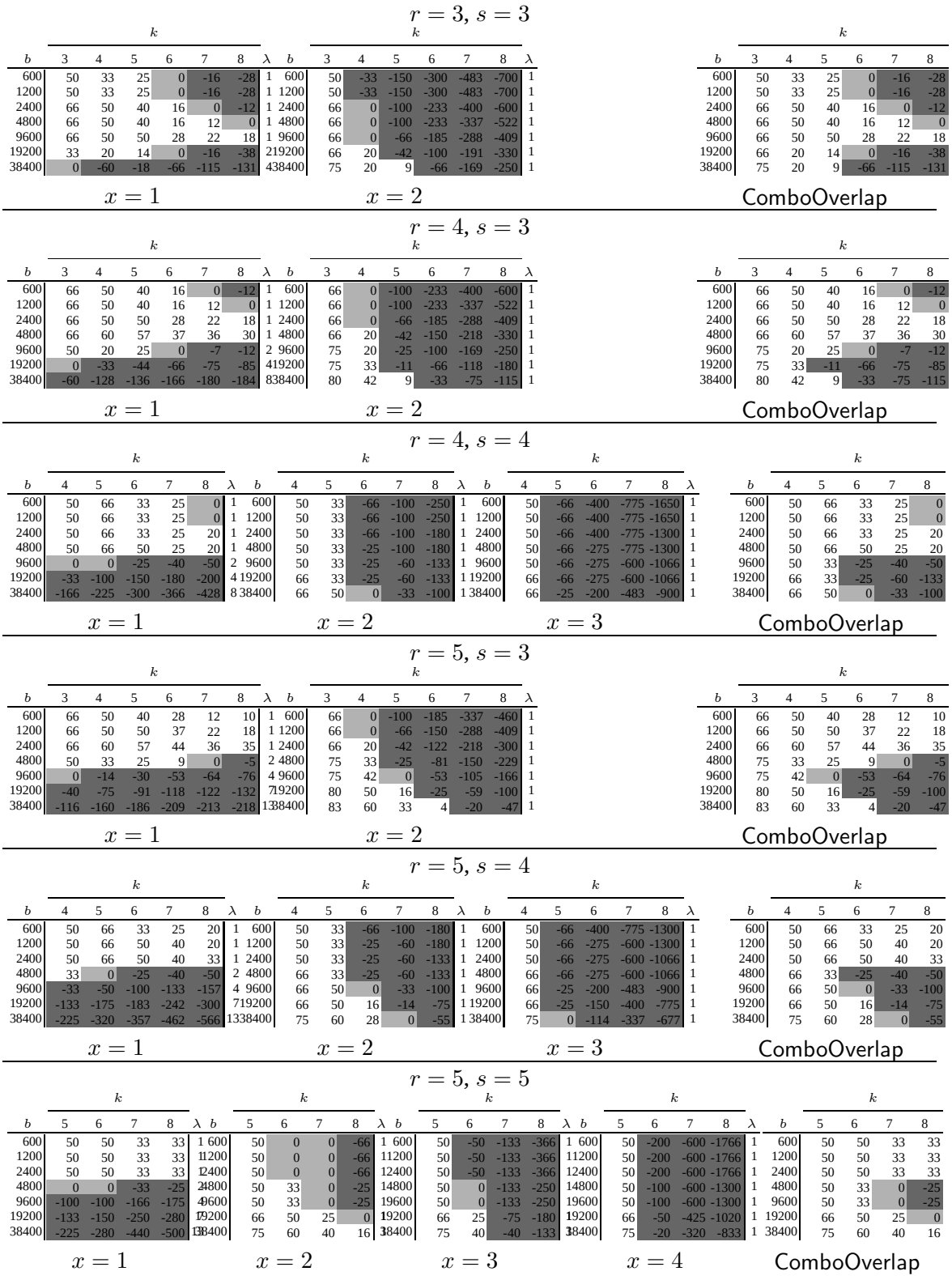
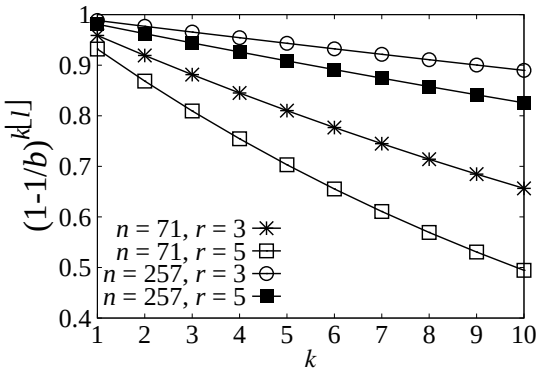
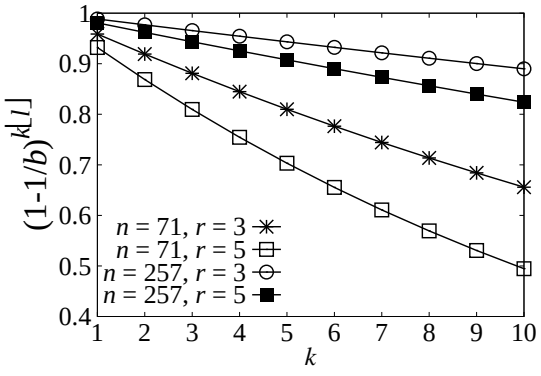


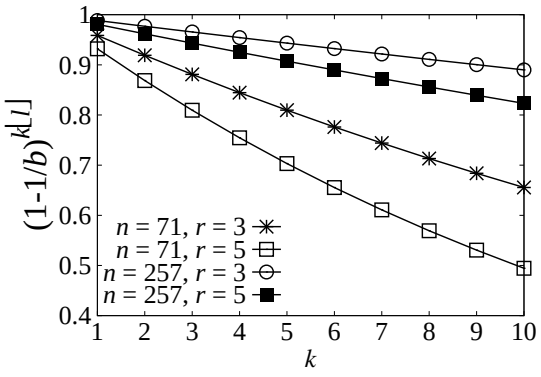
Figure 3.11: $\mathit{lbAvail}^{\mathsf{so}}(x, \lambda) - \mathit{prAvail}^{\mathsf{rnd}}$ for SimpleOverlap(x, λ) placements and $\mathit{lbAvail}^{\mathsf{co}}(\lambda_0, \ldots, \lambda_{s-1}) - \mathit{prAvail}^{\mathsf{rnd}}$ for best ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ placement (right most column) when $s > 2$, as a percentage of the maximum possible improvement $b - \mathit{prAvail}^{\mathsf{rnd}}$, when $n = 257$



$b = 2400$



$b = 9600$



$b = 38400$

Figure 3.12: $\left(1 - \frac{1}{b}\right)^{k\lfloor \ell \rfloor}$ for various n and r, as a function of k

Lemma 8. Suppose $s = 1$, $k < n/2$, and $\ell = \frac{rb}{n}$. Then,

$$\textit{prAvail}^{\,\mathsf{rnd}} \leq b\left(1 - \frac{1}{b}\right)^{k\lfloor\ell\rfloor}$$

Proof. In choosing k nodes to fail, our adversary is guaranteed to be able to fail $k\lfloor\ell\rfloor$ replicas (because $k < n/2$) and, since $s = 1$, every object with one or more replicas in these $k\lfloor\ell\rfloor$ replicas. Let F denote the number of failed objects after the adversary induces k node failures. Contrast this scenario to sampling $k\lfloor\ell\rfloor$ objects with replacement from the objects $\mathsf{obj}_1, \ldots, \mathsf{obj}_b$, and let Y be a random variable capturing the number of *distinct* objects sampled. We claim that $\mathbb{P}(F \geq f) \geq \mathbb{P}(Y \geq f)$ due to two key differences between our adversary's scenario and simply sampling objects at random with replacement: First, since a placement places only one replica per object on any node, once the adversary selects a node to fail, it is *guaranteed* no repetitions of the same object on that node. Second, our adversary can encounter at most r replicas of any object (versus up to $k\lfloor\ell\rfloor$ when sampling objects uniformly at random with replacement). As such,

$$\mathbb{E}(F) = \sum_{f=1}^{\infty}\mathbb{P}(F \geq f) \geq \sum_{f=1}^{\infty}\mathbb{P}(Y \geq f) = \mathbb{E}(Y) = b\left[1 - \left(1 - \frac{1}{b}\right)^{k\lfloor\ell\rfloor}\right]$$

where the last step is well-known (e.g., [33, p. 31]). As such, when $f = b\left[1 - \left(1 - \frac{1}{b}\right)^{k\lfloor\ell\rfloor}\right]$ we have $\textit{Vuln}^{\mathsf{rnd}}(f) \geq 1$, and so $\textit{prAvail}^{\,\mathsf{rnd}} \leq b\left(1 - \frac{1}{b}\right)^{k\lfloor\ell\rfloor}$. □

To see one implication of this lemma, recall that $(1 - \frac{1}{b})^b$ converges to e^{-1} as $b \longrightarrow \infty$. So, for large enough b, $\textit{prAvail}^{\,\mathsf{rnd}}$ is at most approximately $b(e^{-r/n})^k$. In terms of parameter values tested elsewhere in this chapter, Figure 3.12 shows how $\frac{1}{b}\textit{prAvail}^{\,\mathsf{rnd}} = (1 - 1/b)^{k\lfloor\ell\rfloor}$ behaves for small numbers of node failures (c.f., the $s = 1$ case of Figure 3.7). Figure 3.12 is plotted for $b = 2400$, $b = 9600$ and $b = 38400$ (all three are virtually indistinguishable). This graph shows that the availability of Random placements, as a fraction of b, decays essentially linearly in the number k of failed nodes, with a slope that grows smaller as n increases or r decreases (since each node then hosts fewer object replicas).

3.4 Conclusion

In this chapter we explored replica placement strategies based on t-packings, which we here called SimpleOverlap(x, λ) placements, for maximizing the availability of objects in the face of the worst k node failures out of n nodes total. We showed that a SimpleOverlap(x, λ) placement provides availability that is c-competitive with optimal, for a specified constant c (for constant n, k, replicas r, and fatality threshold s). We then devised a placement strategy called ComboOverlap$(\lambda_0, \ldots, \lambda_{s-1})$ that combines multiple SimpleOverlap(x, λ) placements, and a dynamic programming algorithm that selects $\lambda_0, \ldots, \lambda_{s-1}$ so as to maximize (our lower bound on) the availability of the resulting ComboOverlap placement for a chosen k. We showed that a resulting ComboOverlap placement is not particularly sensitive to the value of k with which it is configured, however; for the parameter values we explored, it offers availability for nearby $k' \neq k$ within $\approx 99\%$ of what the best ComboOverlap placement for k' failed nodes would have. Finally, we demonstrated and dissected the improvements offered by ComboOverlap over Random replica placement, based on our analysis of the expected availability supported by Random placement in our worst-case model.

Our algorithms leverage t-packings for parameters for which maximum t-packings (also called t-designs, see Section 3.2.3) are known to exist, meaning that based on current knowledge, realistically our results are limited to $r \leq 5$. Fortunately, this suffices for a wide array of data center applications in practice. Our work does, however, provide further impetus to advance the state-of-the-art in t-packing construction.

CHAPTER 4 CONCLUSION

In this thesis we have identified a previously under-explored opportunity for improving the security of replicated objects in distributed systems, namely the *placement* of their replicas to manage the degree to which objects' replicas reside together on the same physical nodes. Specifically, we have leveraged placements in two novel ways to improve either the confidentiality or the availability of replicated objects.

The first example, presented in Chapter 2, is the system called *StopWatch*, from which an infrastructure-as-a-service cloud can be constructed that convincingly defends its tenant VMs from timing-based side-channel attacks mounted by other tenant VMs. The basic idea behind *StopWatch* is to eliminate independent sources of clocks where possible by making some clock sources functions of others, and then to leverage VM replication and placement to mitigate those independent clock sources that could otherwise not be eliminated — namely those arising from the I/O subsystem (network interrupts, disk interrupts, etc.). Then, by placing each VM's replicas so that sufficiently few overlap with each other VM's replicas, *StopWatch* ensures that no VM observes timings that are substantially influenced by the behavior of any other VM. In particular, this is done by ensuring that any VM's replicas observe event timings that represent the median timing of each event across all of its replicas. In this way, if only a minority of a VM's replicas are co-located with the replicas of any other single VM, the behaviors of each other VM will influence these median timings minimally. We showed that *StopWatch* can accomplish timing side-channel defense in this way while incurring overheads that we believe to be reasonable in light of the strong defenses it provides, including much less than $3\times$ overhead for even I/O intensive applications in our tests.

The second example in this thesis, described in Chapter 3, uses placement of object replicas to improve availability of objects. While enhanced availability is a conventional use of replication, in this chapter we specifically targeted improved availability of objects against an adversary that can intelligently choose which physical nodes to fail (limited only by a budget of nodes it can fail),

in contrast to previous treatments of replica placement to address only probabilistic failures. We showed that in our threat model, careful placement can be used to achieve better object availability than the most popular placement approach under probabilistic node failures, namely random replica placement. The specific placement that we demonstrated to do so involved placing replicas so as to manage the overlaps of different objects' replicas, and to optimally tune those overlaps (using a dynamic programming algorithm) to accommodate the number of object replicas, the number of node failures, the number of object replica failures that disables the object, the number of nodes, and the number of objects.

Recall from our discussion of Chapter 1 that security is typically viewed as addressing confidentiality, availability, or integrity. This thesis has demonstrated that in replicated systems and specific threat models, placement of replicas can be managed in such a way that improves either confidentiality or availability. A natural question, then, is whether placement can be used to effectively improve object integrity. This is undoubtedly true if the compromise of all of one object's replicas by an attacker (e.g., due to a software vulnerability in the object) can be leveraged (e.g., through privilege escalation) to compromise the nodes hosting those replicas and so all replicas that those nodes host. In this case, managing the overlaps of objects' replicas can contain the damage to other objects, particularly if each object leverages Byzantine fault-tolerant replica coordination protocols (e.g., [55, 73, 24, 18, 45]) among its replicas to overcome these compromises. We leave as future work the exploration of other such opportunities for using replica placement to improve facets of security.

BIBLIOGRAPHY

[1] Abel, R. J. R. and Greig, M. (2007). *BIBDs with small block size*, chapter 3. In [19], second edition.

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. (2002). FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation*, pages 1–14.

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

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

[5] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. (2010). Efficient system-enforced deterministic parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation*.

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

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

[8] Bellard, F. (2005). QEMU, a fast and protable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46.

[9] Bernard, S. and Le Fessant, F. (2009). Optimizing peer-to-peer backup using lifetime estimations. In *2009 EDBT/ICDT Workshops*, pages 26–33.

[10] Bhagwan, R., Savage, S., and Voelker, G. M. (2002). Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, Department of Computer Science and Engineering, University of California, San Diego.

[11] Bienia, C. (2011). *Benchmarking modern multiprocessors*. PhD thesis, Princeton University.

[12] Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M. (2000). Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 34–43.

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

[14] Borodin, A. and El-Yaniv, R. (1998). *Online Computation and Competitive Analysis*. Cambridge University Press.

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

[16] Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. In *12th USENIX Security Symposium*, pages 1–14.

[17] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). *The primary-backup approach*, chapter 8. In [63], second edition.

[18] Castro, M. and Liskov, B. (2002). Practical Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 20(4):398–461.

[19] Colbourn, C. J. and Dinitz, J. H., editors (2007). *Handbook of Combinatorial Designs*. Chapman Hall/CRC, second edition.

[20] Colbourn, C. J., Dinitz, J. H., and Stinson, D. R. (1999). Applications of combinatorial designs to communications, cryptography, and networking. In Lamb, J. D. and Preece, D. A., editors, *Surveys in Combinatorics, 1999*, pages 37–100. Cambridge University Press.

[21] Colbourn, C. J. and Mathon, R. (2007). *Steiner systems*, chapter 5. In [19], second edition.

[22] Colbourn, C. J. and Van Oorschot, P. C. (1989). Applications of combinatorial designs in computer science. *ACM Computing Surveys*, 21(2):223–250.

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

[24] Cristian, F., Aghili, H., Strong, R., and Dolev, D. (1995). Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information at Computation*, 118(1):158–179.

[25] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174.

[26] Dasgupta, S., Papadimitriou, C., and Vazirani, U. (2008). *Algorithms*. McGraw-Hill.

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

[28] Deza, E. and Deza, M. (2006). *Dictionary of Distances*. Elsevier.

[29] Dinitz, J. H. and Stinson, D. R. (1992a). *A brief introduction to design theory*, chapter 1. In [30].

[30] Dinitz, J. H. and Stinson, D. R., editors (1992b). *Contemporary Design Theory: A Collection of Surveys*. Wiley-Interscience.

[31] Douceur, J. and Wattenhofer, R. (2001). Competitive hill-climbing strategies for replica placement in a distributed file system. In *15th International Symposium on Distributed Computing*, pages 48–62.

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

[33] Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, Inc., third edition.

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

[35] Gao, D., Reiter, M. K., and Song, D. (2009). Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance. *IEEE Transactions on Dependable and Secure Computing*, 6(2):96–110.

[36] Garcia-Molina, H. and Barbara, D. (1985). How to assign votes in a distributed system. *Journal of the ACM*, 32:841–860.

[37] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google file system. In *19th ACM Symposium on Operating Systems Principles*, pages 29–43.

[38] Gifford, D. K. (1979). Weighted voting for replicated data. In *7th ACM Symposium on Operating System Principles*.

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

[40] Goodrich, M. T. and Tamassia, R. (2011). *Introduction to Computer Security*. Addison-Wesley.

[41] Güngör, M., Bulut, Y., and Çalık, S. (2009). Distributions of order statistics. *Applied Mathemathical Sciences*, 3(16):795–802.

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

[43] Hanani, H., Hartman, A., and Kramer, E. S. (1983). On three-designs of small order. *Discrete Mathematics*, 45(1):75–97.

[44] Hanani, M. (1960). On quadruple systems. *Canad. J. Math.*, 12:145–157.

[45] Hendricks, J., Sinnamohideen, S., Ganger, G. R., and Reiter, M. K. (2010). Zzyzx: Scalable fault tolerance through Byzantine locking. In *40th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 363–372.

[46] Herlihy, M. P. and Tygar, J. D. (1988). How to make replicated data secure. In *Advances in Cryptology – CRYPTO '87 Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 379–391.

[47] Herzberg, A., Shulman, H., Ullrich, J., and Weippl, E. (2013). Cloudoscopy: Services discovery and topology mapping. In *2013 ACM Workshop on Cloud Computing Security*, pages 113–122.

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

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

[50] Intel Manual (2011). *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

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

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

[53] Khosrovshahi, G. B. and Laue, R. (2007). *t-designs with $t \geq 3$*, chapter 4. In [19], second edition.

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

[55] Lamport, L., Shostak, R., and Pease, M. (1989). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[56] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. (1992). Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310.

[57] Li, P., Gao, D., and Reiter, M. K. (2013). Mitigating access-driven timing channels in clouds using StopWatch. In *43rd IEEE/IFIP International Conference on Dependable Systems and Networks*.

[58] Lindner, C. C. and Rodger, C. A. (2008). *Design Theory*, chapter 1. CRC Press.

[59] MacCormick, J., Murphy, N., V.Ramasubramanian, Wieder, U., Yang, J., and Zhou, L. (2009). Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions on Storage*, 4.

[60] Mathon, R. and Rosa, A. (2007). *2-(v, k, λ) designs of small order*, chapter 1. In [19], second edition.

[61] Mills, W. H. and Mullin, R. C. (1992). *Coverings and packings*, chapter 9. In [30].

[62] Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing*. Cambridge University Press.

[63] Mullender, S., editor (1993). *Distributed Systems*. Addison-Wesley, second edition.

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

[65] Ng, W. K. and Ravishankar, C. V. (1995). Coterie templates: A new quorum construction method. In *15th International Conference on Distributed Computing Systems*, pages 92–99.

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

[67] Ogilvy, C. S. (1990). *Excursions in Geometry*, chapter 3-4. Dover.

[68] On, G., Schmitt, J., and Steinmetz, R. (2003). Quality of availability: Replica placement for widely distributed systems. In *11th International Conference on Quality of Service*, pages 325–342.

[69] Ostergard, P. R. and Pottonen, O. (2008). There exists no Steiner system $S(4, 5, 17)$. *Journal of Combinatorial Theory, Series A*, 115(8):1570 – 1573.

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

[71] Raghavarao, D. and Padgett, L. V. (2005). Balanced incomplete block designs — applications. In *Block Designs: Analysis, Combinatorics and Applications*, chapter 5. World Scientific.

[72] Raj, H., Nathuji, R., Singh, A., and England, P. (2009). Resource management for isolation enhanced cloud services. In *ACM Workshop on Cloud Computing Security*, pages 77–84.

[73] Reiter, M. K. (1994). Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *2nd ACM Conference on Computer and Communication Security*, pages 68–80.

[74] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). 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.

[75] Rzadca, K., Datta, A., and Buchegger, S. (2010). Replica placement in P2P storage: Complexity and game theoretic analyses. In *30th IEEE International Conference on Distributed Computing Systems*, pages 588–609.

[76] Santos, J. R., Muntz, R. R., and Ribeiro-Neto, B. (2000). Comparing random data allocation and data striping in multimedia servers. In *2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 44–55.

[77] Schneider, F. B. (1987). Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University.

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

[79] Schneider, F. B. (1993). *What good are models and what models are good?*, chapter 2. In [63], second edition.

[80] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10.

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

[82] Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71.

[83] Turn, R. and Habibi, J. (1986). On the interactions of security and fault-tolerance. In *9th NBS/NCSC National Computer Security Conference*, pages 138–142.

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

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

[86] VMWare, Inc. (2009). Protecting mission-critical workloads with VMware fault tolerance. `http://www.vmware.com/resources/techresources/1094`.

[87] VMWare Information Guide (2010). *Timekeeping in VMware Virtual Machines*. VMWare Inc.

[88] Wray, J. C. (1991). An analysis of covert timing channels. In *1991 IEEE Symposium on Security and Privacy*, pages 2–7.

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

[90] Yin, J., Venkataramani, A., Martin, J.-P., L.Alvisi, and Dahlin, M. (2002). Byzantine fault-tolerant confidentiality. In *International Workshop on Future Directions in Distributed Computing*.

[91] Yu, H. and Gibbons, P. B. (2007). Optimal inter-object correlation when replicating for availability. In *26th ACM Symposium on Principles of Distributed Computing*, pages 254–263.

[92] Yu, H., Gibbons, P. B., and Nath, S. (2006). Availability of multi-object operations. In *3rd USENIX Symposium on Networked Systems Design & Implementation*.

[93] Yu, H. and Vahdat, A. (2002). Minimal replication cost for availability. In *21st ACM Symposium on Principles of Distributed Computing*, pages 98–107.

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

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

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

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