New Cloud Architectures For The Next Generation Internet

A dissertation presented

by

Fangfei Zhou

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

April, 2012

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS COMPLETION APPROVAL FORM

THESIS TITLE:

AUTHOR:

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Ravi Sundaram 4/24/2012.
Thesis Advisor Date

[illegible] 4/24/2012
Thesis Reader *Date*

[illegible] 4/24/2012
Thesis Reader *Date*

[illegible] 4/24/2012
Thesis Reader *Date*

GRADUATE SCHOOL APPROVAL:

[illegible] 4/25/2012
Director, Graduate School *Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[illegible] 4/28/2012
Recipient's Signature *Date*

Distribution: *One Copy to Thesis Advisor*
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)

Abstract

Cloud computing has ushered in a new paradigm with the availability of computing as a service, letting customers share the same physical infrastructure and purchase computing resources on demand (e.g. Amazon EC2 and Windows Azure). The multi-tenancy property of cloud computing offers clients flexibility while creating a unique set of challenges in areas such as reliability and security.

In this thesis we study one challenge (SecureCloud) and three opportunities (DNSCloud, WebCloud and SamaritanCloud). In SecureCloud we explore how multi-tenancy, or the sharing of resources across users, can lead to the undermining of privacy and security. Taking Amazon EC2 as an example we identify an important scheduling vulnerability in Virtual Machine Monitors (VMMs). We create an attack scenario and demonstrate how it can be used to steal cycles in the cloud. We also discuss how attacks can be coordinated across the cloud on a collection of VMs. We present a general framework of solutions to combat such attacks. DNSCloud, WebCloud and SamaritanCloud are proposals for new architectures that improve delivery of existing infrastructural services and enable entirely new functionalities. The Domain Name System (DNS) has long been recognized as the Achilles' heel of the Internet and a variety of new (cache-poisoning and other) attacks surfacing over the past few years have only served to reinforce that notion. We present DNSCloud, a new architecture for providing a more robust DNS service that does not require a forklift upgrade (unlike DNSSEC). Today, content on Web sites like online social networks is created at the edge of network but distributed using a traditional client-server model. WebCloud is a novel cloud architecture that leverages the burgeoning phenomenon of social networks for enabling a more efficient and scalable system for peer-to-peer content delivery. SamaritanCloud is a proposal for a new architecture that exploits the mobility of personal computing devices to share relevant locality-specific information. It allows people to offer physical help to each other remotely, in a secure and private way. Taken as a whole this thesis represents a synthesis of theory and practice that will hasten the ongoing transition to the era of cloud computing.

Acknowledgments

To Master.
To My parents and husband.

I would particularly like to thank my advsior, Prof. Ravi Sundaram, for his academic and life lessons. His thoughtful insights as well as his relentless encouragement, trust in my hard work, and passion for novel work that allowed me to complete my dream of obtaining a Ph.D. I am also very grateful to Prof. Peter Desnoyers, Prof. Alan Mislove and Dr. Brian A. LaMacchia from Microsoft for serving on my committee and for the feedback on my thesis.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Over the past decades, data and applications have been slowly but steadily migrating to the Internet (or its more fashionable synonym, the “Cloud”), with services being delivered to end-users through a client-server model using standardized over-the-web protocols. This outsourcing to the Internet continues to be a work-in-progress with cloud-computing the phrase du jour.

Cloud computing [20] is built on top of server virtualization technology [21], which allows multiple instances of virtual machines running on a single physical server. A cloud computing model is comprised of a front end and a back end. The front end is the interface for users to interact with the system – for example, users can open and edit files through browser, or remotely connect to virtual machines to enjoy applications and other computing resources. The back end is the cloud itself, it maintains data and applications and delivers them to end users as a service through the Internet.

Some of the biggest cloud computing services include Web-based email (e.g. Hotmail, Gmail), social networking (e.g. Facebook, Twitter, LinkedIn), document hosting services, (e.g. Google Docs, Flickr, Picasa, YouTube) and back up services (e.g. Dropbox), etc. Major corporations including Amazon, Google, Microsoft and Oracle have invested in cloud computing services and offer individuals and businesses a range of cloud-based solutions, e.g. Amazon EC2 [1] and Microsoft

Azure [6]. In these services, users rent virtual machines for running their own applications and are charged by the amount of time their virtual machine is running (in hours or months).

Cloud computing is a new and different way to architect and remotely manage computing resources. Therefore, it is important to study both its shortcomings and its advantages which could help resolve current Internet issues. Here are some of the significant advantages.

- *Lower cost:* Cloud computing requires lower cost for data processing when compared with the older model of maintaining software on local machines. The use of the cloud removes the need for purchasing software and hardware and shifts the costs to a pay-per-use model.

- *Mobility and flexibility:* As the service is delivered through the Internet, users can access the resources no matter where they are located. Instant deployment and the pay-as-you-go business model offer users more flexibility than other computing services. Users do not need to worry about application installation and updates.

- *More capacity:* Cloud computing offers virtually limitless storage and computing power, thus enhancing the user's capability beyond the capacity of their local machines.

- *Device independence:* Finally, the ultimate cloud computing advantage is that users are no longer tethered to a single computer or network. Existing applications and data follow users through the cloud.

On the other hand, cloud computing has created its own challenges. One of the biggest concerns about cloud computing is security. Cloud computing also requires a constant-on and high-bandwidth Internet connection, which is a limitation for some clients. As data and application are stored on provider owned machines, clients lose their control over the software and become dependent on the provider to maintain, update and manage it. Cloud computing can also bring risks in the areas of privacy and confidentiality. Clients storing their sensitive data and information on third-party servers may become compromised if the provider has inadequate security in terms of technology or processes.

Today’s cloud computing services are typically categorized into the following three classes of service: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS), each with their own security issues.

- *SaaS:* Cloud infrastructures providing software services often employ a multi-tenancy system architecture, where users access the same environment for different applications. Optimization of speed, security, availability, recovery are required in such services.

- *PaaS:* Cloud infrastructures allowing clients to develop cloud services and applications should provide stable programming environment, tools and configuration management.

- *IaaS:* Cloud infrastructures delivering virtualized resources on demand should meet growing or shrinking resource demand from clients, as well as guarantee fair share among users.

A major shortcoming in the above approaches is the lack of effective and efficient architectures that deliver the much-touted benefits of the cloud while overcoming its shortcomings.

1.2 Overview

This thesis proposes new architectures for cloud-computing that not only fix existing infrastructural deficiencies but also pave the way for newer and improved services. We present four new architectures.

SecureCloud [141]: Theft-free schedulers for multi-tenant architectures. Secure Cloud explores the security issues of cloud infrastructures. In cloud computing services, customers rent virtual machines (VMs) running on hardware owned and managed by third-party providers, and pay for the amount of time their VM is running (in hours or months), rather than by the amount of CPU time used. Under this business model, the choice of scheduling algorithm of the hypervisor involves a trade-off between factors such as fairness, usage caps and scheduling latency. As in operating systems, a hypervisor scheduler may be vulnerable to behavior by virtual machines which results in inaccurate or unfair scheduling. However, cloud computing represents a new environment for such attacks for two reasons. First, the economic model of many services renders them vulnerable to theft-of-service attacks, which can be successful with far lower degrees of unfairness than required for strong denial-of-service attacks. In addition, the lack of detailed application knowledge in the hypervisor (e.g. to differentiate I/O wait from voluntary sleep) makes it more difficult to harden a hypervisor scheduler against malicious behavior. Therefore, verifying the scheduling fairness in cloud computing services is very important to both the providers and the customers. In Chapter 2, we study the scheduling algorithm used by Amazon EC2. We discover that the hypervisor scheduler in Amazon EC2 is vulnerable to behavior by virtual machines which results in inaccurate or unfair scheduling. We describe our attack scenario, provide a novel analysis of the necessary conditions for theft-of-service attacks, and propose scheduler modifications to eliminate the vulnerability. Moreover, our attack itself provides a mechanism for detecting the co-placement of VMs, which in conjunction with appropriate algorithms can be utilized to reveal this mapping. We present an algorithm that is provably optimal when the maximum partition size is bounded. In the unbounded case we show upper and lower bounds using the probabilistic method [18] in conjunction with a sieving technique.

DNSCloud [64]: DNS architecture protects itself from cache-poisoning attacks. DNSCloud explores the domain of new cloud architectures that allow for improved and robust Internet services. The Domain Name System or DNS [85] is a critical and integral part of the Internet. Kaminsky [52, 74, 100] showed that DNS caches throughout the world were susceptible to cache poisoning. This could lead to large-scale impersonation attacks by fake websites. Not only could it potentially lead to the personal and financial ruin of individuals but successful poisoning of this system could also result in catastrophic consequences at a national level [13, 81] were any infrastructural networks such as the military network domains to be targeted by an enemy. Currently, there are only two ways to mitigate this vulnerability, patching the DNS server or patching the DNS protocol itself. Patching the DNS protocol, i.e. substituting DNSSEC [85] instead of DNS requires a forklift upgrade at both the client and resolving name-server ends and for now the world has settled for updating and patching the DNS server software (typically BIND [85]). The problem with the patch however, is that it requires upgrades to all client side resolving name servers, which is the more vulnerable and numerous end of things. In section 3, we propose DNSCloud, a new cloud-based architecture, to dramatically reduce the probability of suffering from a poisoned cache in the DNS.

WebCloud [142]: P2P content delivery system for (wide-area) social networks. Over the past few years, we have witnessed the beginnings of a shift in the patterns of content creation and exchange over the web. Previously, content was primarily created by a small set of entities and was delivered to a large audience of web clients. Now, individual Internet clients are creating content that makes up a significant fraction of Internet traffic [17, 47]. The net result is that, content today is generated by a large number of clients located at the edge of the network, is of more uniform popularity, and exhibits a workload that is governed by the social network. Unfortunately, existing architectures for content distribution are ill-suited for these new patterns of content creation and exchange. In section 4.6.1, we propose WebCloud - a content delivery system designed for social networks. The key insight is to leverage the spatial and temporal locality of interest between social network clients.

SamaritanCloud: Location-based cyber-physical network for obtaining real-world assistance. The tremendous rise in the popularity of social networks has been a defining aspect of the online experience over the last few years. Social network services provide a platform allowing for the building of social relations among people that share similar interests and actives. More recently the phenomenon of geosocial networking has come to the fore. These services use the GPS capabilities of mobile devices to enable additional social dynamics. In geosocial network services such as Foursquare, Gowalla, Facebook Places, and Yelp [3–5, 9], clients share their current locations as well as recommendations for events, food or other interests. In section 5 we propose not just a new architecture, but, in fact, a new service – SamaritanCloud – the goal of which is to provide a way for people to connect with others (possibly strangers) in a remote location and obtain (physical) help from them. It is possible that such a service may never take off but we believe there are several instances in which such a service can be of use to people, e.g., please tell the marathon runner with bib #123 I will wait at the finish line; did I leave my textbook in the restroom? is there a brown puppy roaming in the playground? Such a service will require efficient technical solutions to problems such as scalability, privacy, reputation etc., to overcome the social barriers of soliciting help from strangers. We focus primarily on the technical aspects of scalability – efficiently finding candidates for a request by comparing client profiles with request criteria; efficiency – improving server throughput on high dimensional profiles with locality sensitive hashing; and privacy – matching people with strangers in a secure and private way so that the need for help and the ability, desire to assist are disclosed in a safe and controlled manner.

CHAPTER 2

SecureCloud

2.1 Introduction

Server virtualization [21] enables multiple instances of an operating system and applications (*virtual machines* or VMs) to run on the same physical hardware, as if each were on its own machine. Recently server virtualization has been used to provide so-called *cloud computing* services, in which customers rent virtual machines running on hardware owned and managed by third-party providers. Two such cloud computing services are Amazon's Elastic Compute Cloud (EC2) service and Microsoft Windows Azure Platform; in addition, similar services are offered by a number of web hosting providers (e.g. Rackspace's Rackspace Cloud and ServePath Dedicated Hosting's GoGrid) and referred to as Virtual Private Servers (VPS). In each of these services customers are charged by the amount of time their virtual machine is running (in hours or months), rather than by the amount of CPU time used.

The operation of a hypervisor is in many ways similar to that of an operating system; as an operating system manages access by processes to underlying resources, so too a hypervisor must manage access by multiple virtual machines to a single physical machine. In either case the choice of scheduling algorithm will involve a trade-off between factors such as fairness, usage caps and scheduling latency.

As in operating systems, a hypervisor scheduler may be vulnerable to behavior by virtual machines which results in inaccurate or unfair scheduling. Such anomalies and their potential for malicious use have been recognized in the past in operating systems—McCanne and Torek [88] demonstrate a denial-of-service attack on 4.4BSD, and more recently Tsafrir [130] presents a similar attack against Linux 2.6 which was fixed only recently. Such attacks typically rely on the use of periodic sampling or a low-precision clock to measure CPU usage; like a train passenger hiding whenever the conductor checks tickets, an attacking process ensures it is never scheduled when a scheduling tick occurs.

Cloud computing represents a new environment for such attacks, however, for two reasons. First, the economic model of many services renders them vulnerable to theft-of-service attacks, which can be successful with far lower degrees of unfairness than required for strong denial-of-service attacks. In addition, the lack of detailed application knowledge in the hypervisor (e.g. to differentiate I/O wait from voluntary sleep) makes it more difficult to harden a hypervisor scheduler against malicious behavior.

The scheduler used by the Xen hypervisor (and with modifications by Amazon EC2) is vulnerable to such timing-based manipulation—rather than receiving its fair share of CPU resources, a VM running on unmodified Xen using our attack can obtain up to 98% of total CPU cycles, regardless of the number of other VMs running on the same core. In addition we demonstrate a kernel module allowing unmodified applications to readily obtain 80% of the CPU. The Xen scheduler also supports a non-work-conserving (NWC) mode where each VM's CPU usage is "capped"; in this mode our attack is able to evade its limits and use up to 85% of total CPU cycles. The modified EC2 scheduler uses this to differentiate levels of service; it protects other VMs from our attack, but we still evade utilization limits (typically 40%) and consume up to 85% of CPU cycles.

We give a novel analysis of the conditions which must be present for such attacks to succeed, and present four scheduling modifications which will prevent this attack without sacrificing efficiency, fairness, or I/O responsiveness. We have implemented these algorithms, and present experimental results on Xen 3.2.1.

Chapter outline: The rest of chapter is organized as follows. Section 2.2 provides a brief introduction to VMM architectures, Xen VMM and Amazon EC2 as background. Section 2.3 discusses related work. Section 2.4 describes the details of the Xen Credit scheduler. Section 2.5 explains our attacking scheme and Section 2.7 presents experimental results in the lab as well as on Amazon EC2. Section 2.8 details our scheduling modifications to prevent this attack, and evaluates their performance and overhead. Section 2.9 proofs how our attacking scheme could coordinate attacks, and we conclude in Section 2.10.

2.2 Background

We first provide a brief overview of hardware virtualization technology, and of the Xen hypervisor and Amazon's Elastic Compute Cloud (EC2) service in particular.

2.2.1 Hardware virtualization

Hardware virtualization refers to any system which interposes itself between an operating system and the hardware on which it executes, providing an emulated or *virtualized* view of physical resources. Almost all virtualization systems allow multiple operating system instances to execute simultaneously, each in its own *virtual machine* (VM). In these systems a Virtual Machine Monitor (VMM), also known as a *hypervisor*, is responsible for resource allocation and mediation of hardware access by the various VMs.

Modern hypervisors may be classified by the methods of executing guest OS code without hardware access: (a) binary emulation and translation, (b) paravirtualization, and (c) hardware virtualization support. Binary emulation executes privileged guest code in software, typically with just-in-time translation for speed [15]. Hardware virtualization support [2] in recent x86 CPUs supports a privilege level beyond supervisor mode, used by the hypervisor to control guest

OS execution. Finally, para-virtualization allows the guest OS to execute directly in user mode, but provides a set of *hypercalls*, like system calls in a conventional operating system, which the guest uses to perform privileged functions.

2.2.2 The Xen hypervisor

Xen is an open source VMM for x86/x64 [33]. It introduced para-virtualization on the x86, using it to support virtualization of modified guest operating systems without hardware support (unavailable at the time) or the overhead of binary translation. Above the hypervisor there are one or more virtual machines or *domains* which use hypervisor services to manipulate the virtual CPU and perform I/O.

2.2.3 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 is a commercial service which allows customers to run their own virtual machine instances on Amazon's servers, for a specified price per hour each VM is running. Details of the different instance types currently offered, as well as pricing per instance-hour, are shown in Table 2.1.

Table 2.1: Amazon EC2 Instance Types and Pricing. (Spring 2012. Speed is given in "Amazon EC2 Compute Units".)

Instance Type	Memory	Cores $\times$ speed	\$/Hr
Small	1.7GB	1×1	0.085
Large	7.5	2×2	0.34
X-Large	15	4×2	0.68
Hi-CPU Med.	1.7	2×2.5	0.17
Hi-CPU X-Large	7	8×2.5	0.68

Amazon states that EC2 is powered by "a highly customized version of Xen, taking advantage of virtualization" [11]. The operating systems supported are Linux, OpenSolaris, and Windows Server 2003; Linux instances (and likely OpenSolaris) use Xen's para-virtualized mode, and it is suspected that Windows instances do so as well [25].

2.3 Related work

To the best of our knowledge, our work is the first to show how a deliberately misbehaving VM can unfairly monopolize CPU resources in a virtualized environment, with important implications for Cloud Computing environment. However, the concept of a timing attack long predates computers. Tsafrir *et al.* [130] designed a cheat attack based on a similar scenario in context of processes on the Linux 2.6 scheduler, allowing an attacking process to appear to consume no CPU and receive higher priority. McCanne and Torek [88] present the same cheat attack on 4.4BSD, and develop a uniform randomized sampling clock to estimate CPU utilization. They describe sufficient conditions for this estimate to be accurate, but unlike section 2.8.2 they do not examine conditions for a theft-of-service attack. Cherkasova and Gupta *et al.* [31, 32] have done an extensive performance analysis of scheduling in the Xen VMM. They studied I/O performance for the three schedulers: BVT, SEDF and Credit scheduler. Their work showed that both the CPU scheduling algorithm and the scheduler parameters drastically impact the I/O performance. Furthermore, they stressed that the I/O model on Xen remains an issue in resource allocation and accounting among VMs. Since Domain-0 is indirectly involved in servicing I/O for guest domains, I/O intensive domains may receive excess CPU resources by focusing on the processing resources used by Domain-0 on behalf of I/O bound domains. To tackle this problem, Gupta *et al.* [40] introduced the SEDF-DC scheduler, derived from Xen's SEDF scheduler, that charges guest domains for the time spent in Domain-0 on their behalf. Govindan *et al.* [57] proposed a CPU scheduling algorithm as an extension to Xen's SEDF scheduler that preferentially schedules I/O intensive domains. The key idea behind their algorithm is to count the number of packets flowing into or out of each domain and to schedule the one with highest count that has not yet consumed its entire slice. However, Ongaro *et al.* [101] pointed out that this scheme is problematic when bandwidth-intensive and latency-sensitive domains run concurrently on the same host - the bandwidth-intensive domains are likely to take priority over any latency-sensitive domains with little I/O traffic. They explored the impact of VMM scheduler on I/O performance using multiple guest domains concurrently running different types of applications and evaluated 11 different scheduler configurations within Xen VMM with both the SEDF and Credit schedulers. They also proposed multiple methods to improve I/O performance. Weng *et al.* [134] found from their analysis that Xen's asynchronous CPU scheduling strategy wastes considerable physical CPU time. To fix this problem, they presented a hybrid scheduling framework that groups VMs into high-throughput type and concurrent type and determines processing resource allocation among VMs based on type. In a similar vein Kim *et al.* [77] presented a task-aware VM scheduling mechanism to improve the performance of I/O-bound tasks within domains. Their approach employs gray-box techniques to peer into VMs and identify I/O-bound tasks in mixed workloads. There are some configurations and policies [70, 95, 111] proposed to improve Xen security in other perspectives but not scheduling attacks. Very recently, Ristenpart *et al.* [109] instantiate new VMs on EC2 until one is placed co-resident with the target; they then show that known cross-VM side-channel attacks can extract information from the target. However the side-channel attacks were carried out on carefully controlled machines in the lab, not on EC2, and are likely to be significantly more difficult in practice [125], while, our exploit is directly applicable to EC2. We show that our exploit can be used to infer the mapping of VMs to physical hosts (Partition) and this information can then be used to amplify our scheduling attack into a coordinated siege by a collection of VMs. Although there is significant literature on similar problems for contention resolution in multiple-access channels [59, 60, 79], to the best of our knowledge Partition has not been studied earlier.

2.4 Xen Credit Scheduler analysis

In Xen (and other hypervisors) a single virtual machine consists of one or more virtual CPUs (VCPUs); the goal of the scheduler is to determine which VCPU to execute on each physical CPU (PCPU) at any instant. To do this it must determine which VCPUs are idle and which are active, and then from the active VCPUs choose one for each PCPU.

In a virtual machine, a VCPU is idle when there are no active processes running on it and the scheduler on that VCPU is running its *idle task*. On early systems the idle task would loop forever; on more modern ones it executes a HALT instruction, stopping the CPU in a lower-power state until an interrupt is received. On a fully-virtualized system this HALT traps to the hypervisor and indicates the VCPU is now idle; in a para-virtualized system a direct hypervisor call is used instead. When an exception (e.g. timer or I/O interrupt) arrives, that VCPU becomes active until HALT is invoked again.

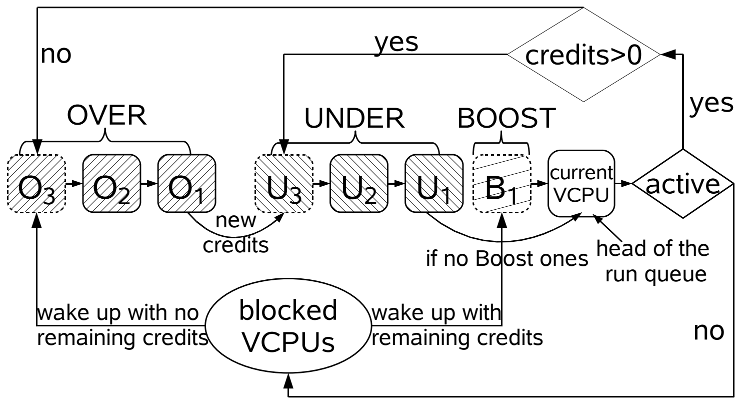
By default Xen uses the Credit scheduler [31], an implementation of the classic *token bucket* algorithm in which credits arrive at a constant rate, are conserved up to a maximum, and are expended during service. Each VCPU receives credits at an administratively determined rate, and a periodic scheduler tick debits credits from the currently running VCPU. If it has no more credits, the next VCPU with available credits is scheduled. Every 3 ticks the scheduler switches to the next runnable VCPU in round-robin fashion, and distributes new credits, capping the credit balance of each VCPU at 300 credits. Detailed parameters (assuming even weights) are:

Fast tick period:	10ms
Slower (rescheduling) tick:	30ms
Credits debited per fast tick:	100
Credit arrivals per fast tick:	100/N
Maximum credits:	300

where N is the number of VCPUs per PCPU. The fast tick decrements the running VCPU by 100 credits every 10 ms, giving each credit a value of 100 μs of CPU time; the cap of 300 credits corresponds to 30 ms, or a full scheduling quantum. Based on their credit balance, VCPUs are divided into three states: *UNDER*, with a positive credit balance, *OVER*, or out of credits, and *BLOCKED* or halted.

The VCPUs on a PCPU are kept in an ordered list, with those in UNDER state ahead of those in OVER state; the VCPU at the head of the queue is selected for execution. In work conserving mode, when no VCPUs are in the UNDER state, one in the OVER state will be chosen, allowing it to receive more than its share of CPU. In non-work-conserving (NWC) mode, the PCPU will go idle instead.

Figure 2.1: Per-PCPU Run Queue Structure



The executing VCPU leaves the run queue head in one of two ways: by going idle, or when removed by the scheduler while it is still active. VCPUs which go idle enter the BLOCKED state and are removed from the queue. Active VCPUs are enqueued after all other VCPUs of the same state—OVER or UNDER—as shown in Figure 2.1.

The basic credit scheduler accurately distributes resources between CPU-intensive workloads, ensuring that a VCPU receiving k credits per 30 ms epoch will receive at least $k/10$ ms of CPU time within a period of 30N ms. This fairness comes at the expense of I/O performance, however, as events such as packet reception may wait as long as 30N ms for their VCPU to be scheduled.

To achieve better I/O latency, the Xen Credit scheduler attempts to prioritize such I/O. When a VCPU sleeps waiting for I/O it will typically have remaining credits; when it wakes with remaining credits it enters the BOOST state and may immediately preempt running or waiting VCPUs with lower priorities. If it goes idle again with remaining credits, it will wake again in BOOST priority at the next I/O event.

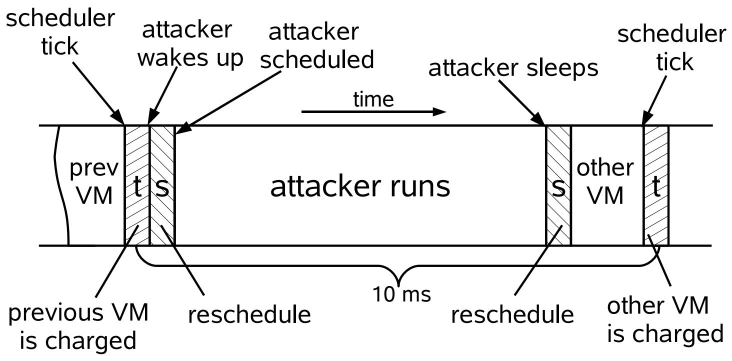
This allows I/O-intensive workloads to achieve very low latency, consuming little CPU and rarely running out of credits, while preserving fair CPU distribution among CPU-bound workloads, which typically utilize all their credits before being preempted. However, as we describe in the following section, it also allows a VM to "steal" more than its fair share of CPU time.

2.5 Anatomy of an attack

Although the Credit scheduler provides fairness and low I/O latency for well-behaved virtual machines, poorly-behaved ones can evade its fairness guarantees. In this section we describe the features of the scheduler which render it vulnerable to attack, formulate an attack scheme, and present results showing successful theft of service both in the lab and in the field on EC2 instances. Our attack relies on periodic sampling as used by the Xen scheduler, and is shown as a timeline in Figure 2.2. Every 10 ms the scheduler tick fires and schedules the attacking VM, which runs for $10 - \varepsilon$ ms and then calls *Halt()* to briefly go idle, ensuring that another VM will be running at the next scheduler tick. Our attack is self-synchronizing due to wake-up after a scheduling tick. In theory the efficiency of this attack increases as ε approaches 0; however in practice some amount of timing jitter is found, and overly small values of ε increase the risk of the VM being found executing when the scheduling tick arrives.

When perfectly executed on the non-BOOST credit scheduler, this ensures that the attacking VM will never have its credit balance debited. If there are N VMs with equal shares, then the N-1 victim VMs will receive credits at a total rate of

Figure 2.2: Attack Timing



$\frac{N-1}{N}$, and will be debited at a total rate of 1.

This vulnerability is due not only to the predictability of the sampling, but to the granularity of the measurement. If the time at which each VM began and finished service were recorded with a clock with the same 10 ms resolution, the attack would still succeed, as the attacker would have a calculated execution time of 0 on transition to the next VM.

This attack is more effective against the actual Xen scheduler because of its BOOST priority mechanism. When the attacking VM yields the CPU, it goes idle and waits for the next timer interrupt. Due to a lack of information at the VM boundary, however, the hypervisor is unable to distinguish between a VM waking after a deliberate sleep period—a non-latency-sensitive event—and one waking for e.g. packet reception. The attacker thus wakes in BOOST priority and is able to preempt the currently running VM, so that it can execute for $10 - \varepsilon$ ms out of every 10 ms scheduler cycle.

2.6 Implementation

2.6.1 User-level

To examine the performance of our attack scenario in practice, we implement it using both user-level and kernel-based code and evaluate them in the lab and on Amazon EC2. In each case we test with two applications: a simple loop we refer to as “Cycle Counter” described below, and the Dhrystone 2.1 [133] CPU benchmark. Our attack described in Section 2.5 requires millisecond-level timing in order to sleep before the debit tick and then wake again at the tick; it performs best either with a tick-less Linux kernel [117] or with the kernel timer frequency set to 1000 Hz.

```
prev=rdtsc()
loop:
   if (rdtsc() - prev) > 9ms
       prev = rdtsc()
       usleep(0.5ms)
```

2.6.2 Kernel-level

To implement kernel-level attack, the most basic way to do is that to add files to the kernel source tree and modify kernel configuration to include the new files in compilation. However, most Unix kernels are monolithic. The kernel itself is a piece of compact code, in which all functions share a common space and are tightly related. When the kernel needs to be updated, all the functions must be relinked and reinstalled and the system rebooted before the change can take affect. This makes modifying kernel level code difficult, especially when the change is made to a kernel which is not in user’s control. Loadable kernel module (LKM) allows developers to make change to kernel when it is running. LKM is an object file that contains code to extend the running kernel of an operating system. LKMs are typically used for one of the three functionalities : device drivers, filesystem drivers and system calls. Some of the advantages of using loadable kernel modules rather than modifying the base kernel are: (1) No kernel recompilation is required. (2) New kernel functionality can be added without root privileges. (3) New functionality takes effect right after module installation, no reboot is required. (4) LKM can be reloaded at run time, it does not add to the size of kernel permanently.

The attack is implemented as a kernel thread which invokes an OS sleep for 10 - ε ms, allowing user applications to run, and then invokes the `SCHED_block` hypercall via the safe halt function. In practice ε must be higher than for the user-mode attack, due to timing granularity and jitter in the kernel.

```
loop:
msleep(8);
safe_halt();
```

In theory the less ε is in kernel module implementation, the more CPU cycles the user application could consume. However due to our evaluation of **msleep**, there is a 1ms timing jitter in kernel. **msleep(8)** allows the kernel thread to wake up on time and temporarily pause all user applications. Then the VM is considered as idle and gets swapped out before debit tick happens. Since **msleep(9)** does not guarantee the thread wake up before the debit tick every time, thus the attacking VM may not be swapped in/swapped out as expected. According to this observation, $\varepsilon = 2$ is a safe choice in implementation for kernel 2.6.

2.7 Evaluation

2.7.1 User-level

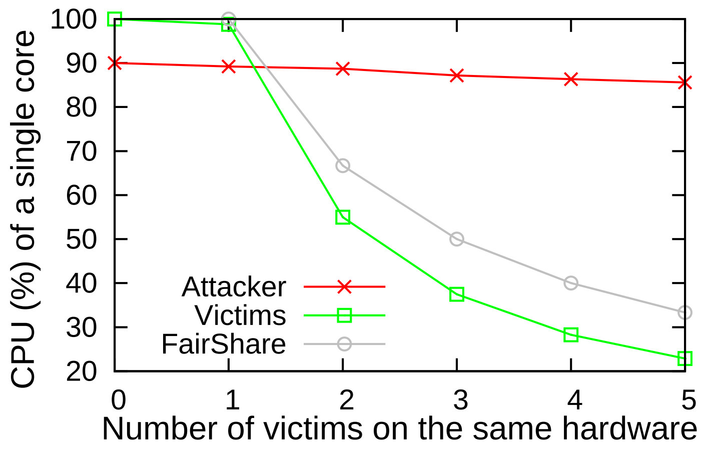
Experiments in the lab

Our first experiments evaluate our attack against unmodified Xen in the lab in work-conserving mode, verifying the ability of the attack to both deny CPU resources to competing "victim" VMs, and to effectively use the "stolen" CPU time for computation. All experiments were performed on Xen 3.2.1, on a 2-core 2.7 GHz Intel Core2 CPU. Virtual machines were 32-bit, para-virtualized, single-VCPU instances with 192 MB memory, each running Suse 11.0 Core kernel 2.6.25 with a 1000 Hz kernel timer frequency. To test our ability to steal CPU resources from other VMs, we implement a "Cycle Counter", which performs no useful work, but rather spins using the RDTSC instruction to read the timestamp register and track the time during which the VM is scheduled. The attack is performed by a variant of this code, "Cycle Stealer", which tracks execution time and sleeps once it has been scheduled for $10 - \varepsilon$ (here ε = 1 ms). Note that the sleep time is slightly less than ε, as the process will be woken at the next OS tick after timer expiration and we wish to avoid over-sleeping. In Figure 2.3a we see attacker and victim performance on our 2-core test system. As the number of victims increases, attacker performance remains almost constant at roughly 90% of a single core, while the victims share the remaining core.

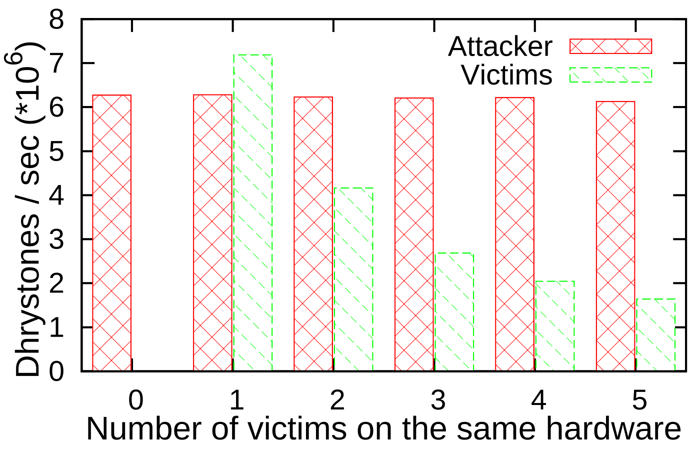
To measure the ability of our attack to effectively use stolen CPU cycles, we embed the attack within the Dhrystone benchmark. By comparing the time required for the attacker and an unmodified VM to complete the same number of Dhrystone iterations, we can determine the *net* amount of work stolen by the attacker. Our baseline measurement was made with one VM running unmodified Dhrystone, with no competing usage of the system; it completed 1.5×10^9 iterations in 208.8 seconds. When running 6 unmodified instances, three for each core, each completed the same 1.5×10^9 iterations in 640.8 seconds on average—32.6% the baseline speed, or close to the expected fair share performance of 33.3%. With one modified attacker instance competing against 5 unmodified victims, the attacker completed in 245.3 seconds, running at a speed of 85.3% of baseline, rather than 33.3%, with a corresponding decrease in victim performance. Full results for experiments with 0 to 5 unmodified victims and the modified Dhrystone attacker are shown in Figure 2.3b. In the modified Dhrystone attacker the TSC register is sampled once for each iteration of a particular loop, as described in the appendix; if this

Figure 2.3: Lab experiments (User Level) - CPU and application performance for attacker and victims.

(a) CPU (cycle stealer)



(b) Application (Dhrystone)



sampling occurs too slowly the resulting timing inaccuracy might affect results. To determine whether this might occur, lengths of the compute and sleep phases of the attack were measured. Almost all (98.8%) of the compute intervals were found to lie within the bounds 9 ± 0.037 ms, indicating that the Dhrystone attack was able to attain timing precision comparable to that of Cycle Stealer. As described in Section 2.5, the attacker runs for a period of length $10 - \varepsilon$ ms and then briefly goes to sleep to avoid the sampling tick. A smaller value of ε increases the CPU time stolen by the attacker; however, too small an ε increases the chance of being charged due to timing jitter. To examine this trade-off we tested values of $10 - \varepsilon$ between 7 and 9.9 ms. Figure 2.7 shows that under lab conditions the peak value was 98% with an execution time of 9.8 ms and a requested sleep time of 0.1 ms. When execution time exceeded 9.8 ms the attacker was seen by sampling interrupts with high probability. In this case it received only about 21% of one core, or even less than the fair share of 33.3%.

Experiments on Amazon

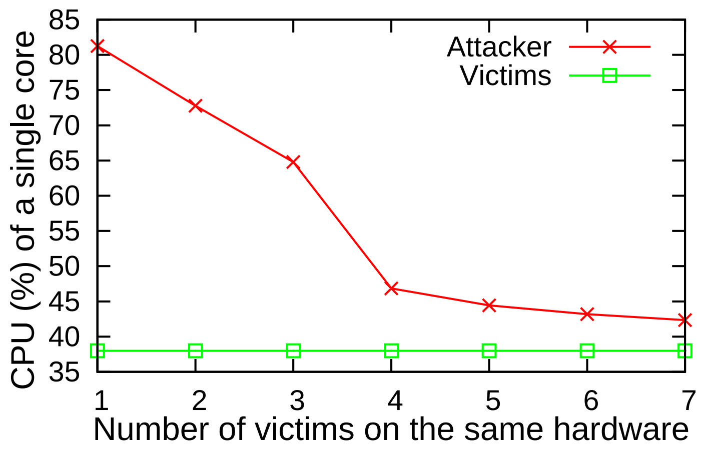
We evaluate our attacking using Amazon EC2 Small instances with the following attributes: 32-bit, 1.7 GB memory, 1 VCPU, running Amazon's Fedora Core 8 kernel $2.6.18$, with a 1000 Hz kernel timer. We note that the VCPU provided to the Small instance is described as having "1 EC2 Compute Unit", while the VCPUs for larger and more expensive instances are described as having 2 or 2.5 compute units; this indicates that the scheduler is being used in non-work-conserving mode to throttle Small instances. To verify this hypothesis, we ran Cycle Stealer in measurement (i.e. non-attacking) mode on multiple Small instances, verifying that these instances are capped to less than $\frac{1}{2}$ of a single CPU core—in particular, approximately 38% on the measured systems. We believe that the nominal CPU cap for 1-unit instances on the measured hardware is 40%, corresponding to an unthrottled capacity of 2.5 units. Additional experiments were performed on a set of 8 Small instances co-located on a single 4-core 2.6 GHz physical system provided by our partners at Amazon.[1] The Cycle Stealer and Dhrystone attacks measured in the lab were performed in this configuration, and results are shown in Figure 2.4a and Figure 2.4b, respectively. We find that our attack is able to evade the CPU cap of 40% imposed by EC2 on Small instances, obtaining up to 85% of one core in the absence of competing VMs. When co-located CPU-hungry "victim" VMs were present, however, EC2 performance diverged from that of unmodified Xen. As seen in Figures 2.4a and 2.4b, co-located VM performance was virtually unaffected by

[1] This configuration allowed direct measurement of attack impact on co-located "victim" VMs, as well as eliminating the possibility of degrading performance of other EC2 customers.

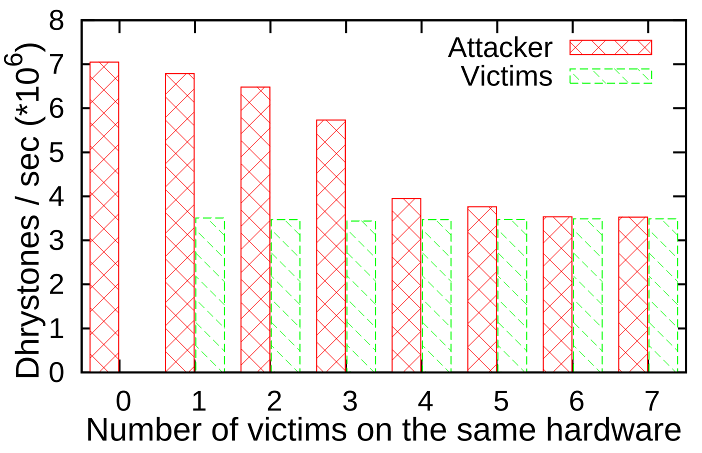
our attack. Although this attack was able to steal cycles from EC2, it was unable to steal cycles from other EC2 customers.

Figure 2.4: Amazon EC2 experiments - CPU and application performance for attacker and victims.

(a) CPU (cycle stealer)



(b) Application (Dhrystone)



2.7.2 Kernel-level

Lab experiments were performed with one attacking VM, which loads the kernel module, and up to 5 victims running a simple CPU-bound loop. In this case the fair CPU share for each guest instance on our 2-core test system would be $\frac{200}{N}\%$ of a single core, where N is the total number of VMs. Due to the granularity of kernel timekeeping, requiring a larger ε, the efficiency of the kernel-mode attack is slightly lower than that of the user-mode attack. We evaluate the kernel level attacking with Cycle Stealer in measurement (non-attacking) mode and unmodified Dhrystone. In our tests, the attacker consumed up to 80.0% of a single core, with the remaining CPU time shared among the victim instances; results are shown in Figure 2.5a. The average amount of CPU stolen by the attacker decreases slightly (from 80.0% to 78.2%) as the number of victims increases; we speculate that this may be due to increased timing jitter causing the attacker to occasionally be charged by the sampling tick. The current implementation of the kernel module does not succeed in stealing cycles on Amazon EC2. We dig into Amazon EC2 kernel synchronization and our analysis of timing traces indicates a lack of synchronization of the attacker to the hypervisor tick, as seen for NWC mode in the lab, below; in this case, however, synchronization was never achieved. The user-level attack displays strong self-synchronizing behavior, aligning almost immediately to the hypervisor tick; we are investigating approaches to similarly strengthen self-synchronizing in the kernel module.

Table 2.2: Lab: Attack performance in non-work-conserving mode with 33.3% limit.

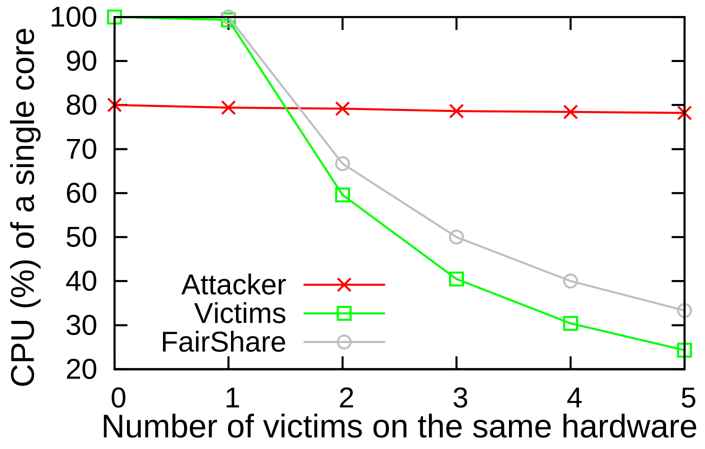
	Cycle Stealer (% of 1 core achieved)	Dhrystones per second	% of baseline
attacker	81.0	5749039	80.0
victims	23.4	1658553	23.1

2.7.3 Non-work conserving mode

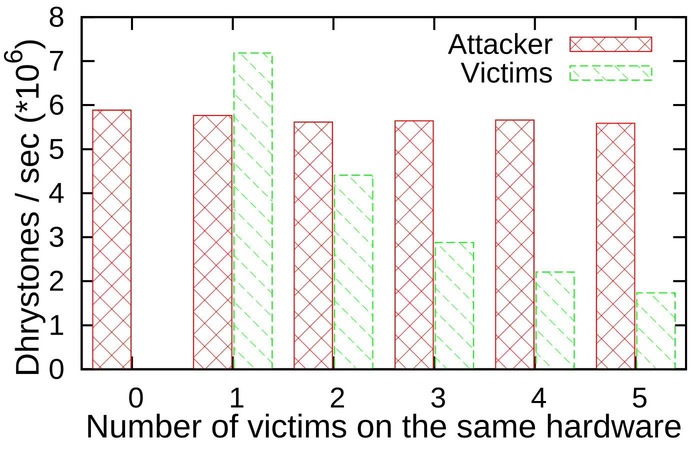
As with most hypervisors, Xen CPU scheduling can be configured in two modes: work-conserving mode and non-work-conserving mode [42]. In order to optimize scheduling and allow near native performance, work-conserving scheduling schemes are efficient and do not waste any processing cycles. As long as there are instructions to be executed and there is enough CPU capacity, work-conserving schemes assign instructions to physical CPU to be carried out. Otherwise the in-

Figure 2.5: Lab experiments (Kernel Level) - CPU and application performance for attacker and victims.

(a) CPU (Cycle Counter)



(b) Application (Dhrystone)



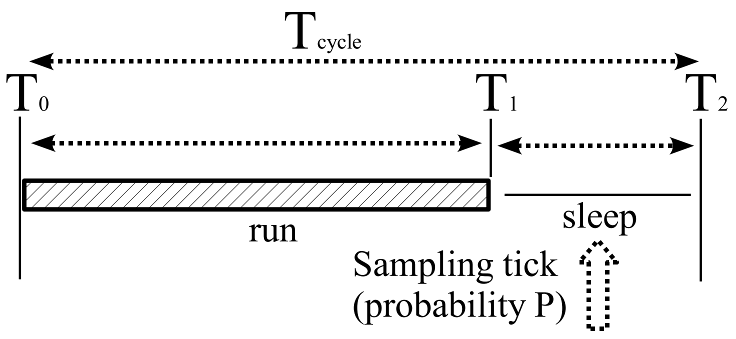
structions will be queued and will be executed based on their priority. In contrast, non-work-conserving schemes allow CPU resources to go unused. In such schemes, there is no advantage to execute an instruction sooner. Usually, the CPU resources are allocated to VMs in proportion to their weights, e.g. two VMs with equal weight will own 50% each of CPU. When one VM goes idle, instead of obtaining the free cycles, the other VM is capped to its 50% sharing. Our attack program helps a VM to evade its capacity cap and obtain more, no matter if other VMs are busy or idle. Additional experiments were performed to examine our attack performance against the Xen scheduler in non-work-conserving mode. One attacker and 5 victims were run on our 2-core test system, with a CPU cap of 33% set for each VM; results are presented in Table 2.2 for both Cycle Stealer and Dhrystone attackers, including user-level and kernel-level implementation. With a per-VM CPU cap of 33%, the attacker was able to obtain 80% of a single core, as well. In each case the primary limitation on attack efficiency is the granularity of kernel timekeeping; we speculate that by more directly manipulating the hypervisor timer it would be possible to increase efficiency. In non-work-conserving case, when a virtual machine running attacking program yields CPU, there is a chance (based on our experiment results, not often though) that reschedule does not happen, the attacking VM is still considered as the scheduled VM when debit tick happens and gets debited. In other words, non-work-conserving mode does not stop our attack, but adds some interferences. In addition we note that it often appeared to take seconds or longer for the attacker to synchronize with the hypervisor tick and evade resource caps, while the user-level attack succeeds immediately.

2.8 Theft-resistant Schedulers

The class of theft-of-service attacks on schedulers which we describe is based on a process or virtual machine voluntarily sleeping when it could have otherwise remained scheduled. As seen in Figure 2.6, this involves a tradeoff—the attack will only succeed if the expected benefit of sleeping for T_{sleep} is greater than the guaranteed cost of yielding the CPU for that time period. If the scheduler is attempting to provide each user with its fair share based on measured usage, then sleeping for a duration t must reduce measured usage by more than t in order to be effective. Conversely, a scheduler which ensures that yielding the CPU will never reduce measured usage more than the sleep period itself will be resistant to such attacks.

This is a broader condition than that of maintaining an unbiased estimate of CPU usage, which is examined by McCanne and Torek [88]. Some theft-resistant schedulers, for instance, may over-estimate the CPU usage of attackers and give

Figure 2.6: Attacking trade-offs. The benefit of avoiding sampling with probability P must outweigh the cost of forgoing T_{sleep} CPU cycles.



them less than their fair share. In addition, for schedulers which do not meet our criteria, if we can bound the ratio of sleep time to measurement error, then we can establish bounds on the effectiveness of a timing-based theft-of-service attack.

2.8.1 Exact scheduler

The most direct solution is the *Exact scheduler*: using a high-precision clock (in particular, the TSC) to measure actual CPU usage when a scheduler tick occurs or when a VCPU yields the CPU and goes idle, thus ensuring that an attacking VM is always charged for exactly the CPU time it has consumed. In particular, this involves adding logic to the Xen scheduler to record a high-precision timestamp when a VM begins executing, and then calculate the duration of execution when it yields the CPU. This is similar to the approach taken in e.g. the recent tickless Linux kernel [117], where timing is handled by a variable interval timer set to fire when the next event is due rather than using a fixed-period timer tick.[2]

2.8.2 Randomized schedulers

An alternative to precise measurement is to sample as before, but on a random schedule. If this schedule is uncorrelated with the timing of an attacker, then over

[2]Although Kim et al. [77] use TSC-based timing measurements in their modifications to the Xen scheduler, they do not address theft-of-service vulnerabilities.

sufficiently long time periods we will be able to estimate the attacker's CPU usage accurately, and thus prevent attack. Assuming a fixed charge per sample, and an attack pattern with period T_{cycle}, the probability P of the sampling timer falling during the sleep period must be no greater than the fraction of the cycle $\frac{T_{sleep}}{T_{cycle}}$ which it represents.

Poisson Scheduler: This leads to a Poisson arrival process for sampling, where the expected number of samples during an interval is exactly proportional to its duration, regardless of prior history. This leads to an exponential arrival time distribution,

$$\Delta T = \frac{-ln\boldsymbol{U}}{\lambda}$$

where *U* is uniform on (0,1) and λ is the rate parameter of the distribution. We approximate such Poisson arrivals by choosing the inter-arrival time according to a truncated exponential distribution, with a maximum of 30 ms and a mean of 10 ms, allowing us to retain the existing credit scheduler structure. Due to the possibility of multiple sampling points within a 10 ms period we use a separate interrupt for sampling, rather than re-using or modifying the existing Xen 10 ms interrupt.

Bernoulli Scheduler: The discrete-time analog of the Poisson process, the *Bernoulli* process, may be used as an approximation of Poisson sampling. Here we divide time into discrete intervals, sampling at any interval with probability *p* and skipping it with probability *q* = *1-p*. We have implemented a Bernoulli scheduler with a time interval of 1 ms, sampling with $p = \frac{1}{10}$, or one sample per 10 ms, for consistency with the unmodified Xen Credit scheduler. Rather than generate a timer interrupt with its associated overhead every 1 ms, we use the same implementation strategy as for the Poisson scheduler, generating an inter-arrival time variate and then setting an interrupt to fire after that interval expires. By quantizing time at a 1 ms granularity, our Bernoulli scheduler leaves a small vulnerability, as an attacker may avoid being charged during any 1 ms interval by sleeping before the end of the interval. Assuming that (as in Xen) it will not resume until the beginning of the next 10 ms period, this limits an attacker to gaining no more than 1 ms every 10 ms above its fair share, a relatively insignificant theft of service.

Uniform Scheduler: The final randomized scheduler we propose is the *Uniform* scheduler, which distributes its sampling uniformly across 10 ms scheduling intervals. Rather than generating additional interrupts, or modifying the time at which the existing scheduler interrupt fires, we perform sampling within the virtual machine switch code as we did for the exact scheduler. In particular, at the beginning of each 10 ms interval (time t_0) we generate a random offset Δ uniformly distributed between 0 and 10 ms. At each VCPU switch, as well as at the 10 ms tick, we check to see whether the current time has exceeded $t_0 + \Delta$. If so, then we debit the currently running VCPU, as it was executing when the "virtual interrupt" fired at $t_0 + \Delta$. Although in this case the sampling distribution is not memoryless, it is still sufficient to thwart our attacker. We assume that sampling is undetectable by the attacker, as it causes only a brief interruption indistinguishable from other asynchronous events such as network interrupts. In this case, as with Poisson arrivals the expected number of samples within any interval in a 10 ms period is exactly proportional to the duration of the interval. Our implementation of the uniform scheduler quantizes Δ with 1 ms granularity, leaving a small vulnerability as described in the case of the Bernoulli scheduler. As in that case, however, the vulnerability is small enough that it may be ignored. We note also that this scheduler is not theft-proof if the attacker is able to observe the sampling process. If we reach the 5 ms point without being sampled, for instance, the probability of being charged 10 ms in the remaining 5 ms is 1, while avoiding that charge would only cost 5 ms.

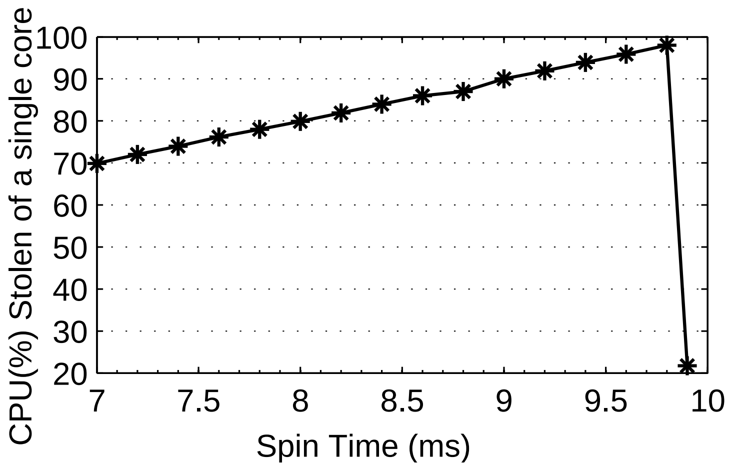
2.8.3 Evaluation

We have implemented each of the four modified schedulers on Xen 3.2.1. Since the basic credit and priority boosting mechanisms have not been modified from the original scheduler, our modified schedulers should retain the same fairness and I/O performance properties of the original in the face of well-behaved applications. To verify performance in the face of ill-behaved applications we tested attack performance against the new schedulers; in addition measuring overhead and I/O performance.

Performance against attack

In Table 2.3 we see the performance of our Cycle Stealer on the Xen Credit scheduler and the modified schedulers. All four of the schedulers were successful in thwarting the attack: when co-located with 5 victim VMs on 2 cores on the unmodified scheduler, the attacker was able to consume 85.6% of a single-CPU with user-level attacking and 80% with kernel-level attacking , but no more than its fair share on each of the modified ones. (Note that the 85.6% consumption with user-level attacking in the unmodified case was limited by the choice of $\varepsilon = 1$ ms, and can increase with suitably reduced values of ε as shown in Figure 2.7.)

Figure 2.7: Lab: User-level attack performance vs. execute times. (note - sleep time $\leq 10-$spin time)



In Table 2.4 we see similar results for the modified Dhrystone attacker. Compared to the baseline, the unmodified scheduler allows the attacker to steal about 85.3% CPU cycles with user-level attacking and 77.8% with kernel-level attacking; while each of the improved schedulers limits the attacker to approximately its fair share.

Table 2.3: Performance of the schedulers against cycle stealer

	CPU(%) obtained by the attacker	
Scheduler	(user-level)	(kernel-level)
Xen Credit	85.6	78.8
Exact	32.9	33.1
Uniform	33.1	33.3
Poisson	33.0	33.2
Bernoulli	33.1	33.2

Table 2.4: Performance of the schedulers against Dhrystone

	CPU(%) obtained by the attacker	
Scheduler	(user-level)	(kernel-level)
Xen Credit	85.3	77.8
Exact	32.2	33.0
Uniform	33.0	33.4
Poisson	32.4	33.1
Bernoulli	32.5	33.3

Overhead measurement

To quantify the impact of our scheduler modifications on normal execution (i.e. in the absence of attacks) we performed a series of measurements to determine whether application or I/O performance had been degraded by our changes. Since the primary modifications made were to interrupt-driven accounting logic in the Xen scheduler, we examined overhead by measuring performance of a CPU-bound application (unmodified Dhrystone) on Xen while using the different scheduler. To reduce variance between measurements (e.g. due to differing cache line alignment [97]) all schedulers were compiled into the same binary image, and the desired scheduler selected via a global configuration variable set at boot or compile time.

Our modifications added overhead in the form of additional interrupts and/or accounting code to the scheduler, but also eliminated other accounting code which had performed equivalent functions. To isolate the effect of new code from that of the removal of existing code, we also measured versions of the Poisson and

Bernoulli schedulers (Poisson-2 and Bernoulli-2 below) which performed all accounting calculations of both schedulers, discarding the output of the original scheduler calculations.

Results from 100 application runs for each scheduler are shown in Table 2.5. Overhead of our modified schedulers is seen to be low—well under 1%—and in the case of the Bernoulli and Poisson schedulers is negligible. Performance of the Poisson and Bernoulli schedulers was unexpected, as each incurs an additional 100 interrupts per second; the overhead of these interrupts appears to be comparable to or less than the accounting code which we were able to remove in each case. We note that these experiments were performed with Xen running in para-virtualized mode; the relative cost of accounting code and interrupts may be different when using hardware virtualization.

We analyzed the new schedulers' I/O performances by testing the I/O latency between two VMs in two configurations. In configuration 1, two VMs executed on the same core with no other VMs active, while in configuration 2 a CPU-bound VM was added on the other core. From the first test, we expected to see the performance of well-behaved I/O intensive applications on different schedulers; from the second one, we expected to see that the new schedulers retain the priority boosting mechanism.

In Table 2.6 we see the results of these measurements. Differences in performance were minor, and as may be seen by the overlapping confidence intervals, were not statistically significant.

Table 2.5: Scheduler CPU overhead, 100 data points per scheduler.

Scheduler	CPU overhead (%)	95% CI
Exact	0.50	0.24 – 0.76
Uniform	0.44	0.27 – 0.61
Poisson	0.04	-0.17 – 0.24
Bernoulli	-0.10	-0.34 – 0.15
Poisson-2	0.79	0.60 – 0.98
Bernoulli-2	0.79	0.58 – 1.00

Table 2.6: I/O latency by scheduler, with 95% confidence intervals.

	Round-trip delay (μs)	
Scheduler	(config. 1)	(config. 2)
Unmodified Xen Credit	53 ± 0.66	96 ± 1.92
Exact	55 ± 0.61	97 ± 1.53
Uniform	54 ± 0.66	96 ± 1.40
Poisson	53 ± 0.66	96 ± 1.40
Bernoulli	54 ± 0.75	97 ± 1.49

2.8.4 Additional discussion

A comprehensive comparison of our proposed schedulers is shown in Table 2.7. The *Poisson* scheduler seems to be the best option in practice, as it has no performance overhead nor vulnerability. Even though it has short-period variance, it guarantees exactly fair share in the long run. The *Bernoulli* scheduler would be an alternative if the vulnerability of up to 1ms is not a concern. The *Uniform* scheduler has similar performance to the Bernoulli one, and the implementation of sampling is simpler, but it has more overhead than Poisson and Bernoulli. Lastly, the *Exact* scheduler is the most straight-forward strategy to prevent cycle stealing, with a relatively trivial implementation but somewhat higher overhead.

Table 2.7: Comparison of the new schedulers

Schedulers	Short-run fairness	Long-run fairness	Low overhead	Ease of implementation	Determi-nistic	Theft-proof
Exact	✓	✓		✓	✓	✓
Uniform		✓		✓		
Poisson		✓	✓			✓
Bernoulli		✓	✓			

2.9 Coordinated cloud attacks

We have shown how an old vulnerability has manifested itself in the modern context of virtualization. Our attack [141] enables an attacking VM to steal cycles by gaming a vulnerability in the hypervisor's scheduler. We now explain how, given a collection of VMs in a cloud, attacks may be coordinated so as to steal the maximal number of cycles.

Consider a customer of a cloud service provider with a collection of VMs. Their goal is to extract the maximal number of cycles possible for executing their computational requirements. Note that the goal of the customer is not to inflict the highest load on the cloud but to extract the maximum amount of useful work for themselves. To this end they wish to steal as many cycles as possible. As has already been explained attacking induces an overhead and having multiple VMs in attack mode on the same physical host leads to higher total overhead reducing the total amount of useful work. Ideally, therefore, the customer would like to have exactly one VM in attack mode per physical host with the other VMs functioning normally in non-attack mode.

Unfortunately, cloud providers such as Amazon's EC2 not only control the mapping of VMs to physical hosts but also withhold information about the mapping from their customers. As infrastructure providers this is the right thing for them to do. By doing so they make it harder for malicious customers to snoop on others. [109] show the possibility of targeting victim VMs by mapping the internal cloud infrastructure though this is much harder to implement successfully in the wild [125]. Interestingly, though our attack can be turned on its head not just to steal cycles but also to identify co-placement. Assume without loss of generality for the purposes of the rest of this discussion that hosts are single core machines. Recall that a regular (i.e. non-attacking) VM on a single host is capped at 38% whereas an attacking VM obtains 87% of the cycles. Now, if we have two attacking VMs on the same host then they obtain about 42% each. Thus by exclusively (i.e without running any other VMs) running a pair of VMs in attack mode one can determine whether they are on the same physical host or not (depending on whether they get 42% or 87%). Thus a customer could potentially run every pair of VMs and determine which VMs are on the same physical host. (Of course, this is under the not-unreasonable assumption that only the VMs of this particular customer can be in attack mode.) Note that if there are n VMs then this scheme requires $\binom{n}{2}$ tests for each pair. This leads to a natural question: what is the most efficient way to discover the mapping of VMs to physical host? This problem has a very clean and beautiful formulation.

Observe that the VMs get partitioned among (an unknown number of) the physical hosts. And we have the flexibility to activate some subset of the VMs in attack mode. We assume (as is the case of Amazon's EC2) that there is no advantage to running any of the other VMs in normal (non-attack) mode since they get their fair share (recall that in EC2 attackers only get additional *spare* cycles). When we activate a subset of the VMs in attack mode then we get back one bit of information for each VM in the subset - namely, whether that VM is the only VM from the subset on its physical host or whether there are 2 or more VMs from the subset on the same host. Thus the question now becomes: what is the fastest way to discover the unknown partition (of VMs among physical hosts)? We think of each subset that we activate as a query that takes unit time and we wish to use the fewest number of queries. More formally:

Partition. You are given an unknown partition $\mathfrak{P} = \{S_1, S_2, \ldots, S_k\}$ of a ground set $[1 \ldots n]$. You are allowed to ask queries of this partition. Each query is a subset $Q = \{q_1, q_2, \ldots, \} \subset [1 \ldots n]$. When you ask the query Q you get back $|Q|$ bits, a bit b_i for each element $q_i \in Q$; let S_{q_i} denote the set of the partition $\mathfrak{P}$ containing q_i; then $b_i = 1$ if $|Q \bigcap S_{q_i}| = 1$ (and otherwise, $b_i = 0$ if $|Q \bigcap S_{q_i}| \geq 2$). The goal is to determine the query complexity of Partition, i.e., you have to find $\mathfrak{P}$ using the fewest queries.

Recall that a partition is a collection of disjoint subsets whose union is the ground set, i.e., $\forall 1 \leq i, j, \leq k, S_i \cap S_j = \emptyset$ and $\bigcup_{i=1}^{k} S_i = [1 \ldots n]$. Observe that one can define both adaptive (where future queries are dependent on past answers) and oblivious variants of Partition. Obviously, adaptive queries have at least as much power as oblivious queries. In the general case we are able to show nearly tight bounds:

Theorem 1. *The oblivious query complexity of* Partition *is* $O(n^{\frac{3}{2}}(\log n)^{\frac{1}{4}})$ *and the adaptive query complexity of* Partition *is* $\Omega(\frac{n}{\log^2 n})$.

Proof Sketch: The upper bound involves the use of the probabilistic method in conjunction with sieving [18]. We are able to derandomize the upper bound to produce deterministic queries, employing expander graphs and pessimal estimators. The lower bound uses an adversarial argument based on the probabilistic method as well. □

In practice, partitions cannot be arbitrary as there is a bound on the number of VMs that a cloud service provider will map to a single host. This leads to the problem B-Partition where all the sets in the partition have a size at most B. In the special case we are able to prove tight bounds:

Theorem 2. *The query complexity of* B-*Partition is* $\theta(\log n)$.

Proof Sketch: The lower bound follows directly from the information-theoretic argument that there are $\Omega(2^{n \log n})$ partitions while each query returns only n bits of information. The upper bound involves use of the probabilistic method (though the argument is simpler than for the general case) and can be derandomized to provide deterministic constructions of the queries. □

Proof Sketch: We prove an upper bound of $O(\log n)$ on the query complexity of B-Partition, where the size of any set in the partition is at most B. Due to constraints of space we exhibit a randomized construction and leave the details of a deterministic construction to [141]. First, we query the entire ground set $[1 \ldots n]$ and this allows us to identify any singleton sets in the partition. So now we assume that every set in our partitions has size at least 2. Consider a pair of elements x and y belonging to different sets S_x and S_y. Fix any other element $y' \in S_y$, arbitrarily (note that such a y' exists because we assume there are no singleton sets left). A query Q is said to be a *separating witness* for the tuple $\mathcal{T} =< x, y, y', S_x >$ iff $Q \bigcap S_x = x$ and $y, y' \in Q$. (Observe that for any such query it will be the case that $b_x = 1$ while $b_y = 0$, hence the term "separating witness". Observe that any partition $\mathfrak{P}$ is completely determined by a set of queries with separating witnesses for all the tuples the partition contains. Now, form a query Q by picking each element independently and uniformly with probability $\frac{1}{2}$. Consider any particular tuple $\mathbb{T} =< x, y, y', S_x >$. Then

$$Pr_Q(Q \text{ is a separating witness for } \mathbb{T}) \geq \frac{1}{2^{B+2}}$$

Recall that $|S_x| \leq B$ since we are only considering partitions whose set sizes are at most B. Now consider a collection $\mathcal{Q}$ of such queries each chosen independently of the other. Then for a given tuple $\mathbb{T}$ we have that

$$Pr_{\mathcal{Q}}(\forall_{Q \in \mathcal{Q}} Q \text{ is not a separating witness for } \mathbb{T}) \leq (1 - \frac{1}{2^{B+2}})^{|\mathcal{Q}|}$$

But there are at most $\binom{n}{B+2} * B^3 \leq n^{B+2}$ such tuples. Hence,

$$Pr_{\mathcal{Q}}(\exists_{\text{tuple } \mathbb{T}} \forall_{Q \in \mathcal{Q}} Q \text{ is not a separating witness for } \mathbb{T}) \leq$$

$$n^{B+2} * (1 - \frac{1}{2^{B+2}})^{|\mathcal{Q}|}$$

Thus by choosing $|\mathcal{Q}| = O((B+2) * 2^{B+2} * \log n) = O(\log n)$ queries we can ensure that the above probability is below 1, which means that the collection $\mathcal{Q}$ contains a separating witness for every possible tuple. This shows that the query complexity is $O(\log n)$. $\square$

2.10 Conclusion

Scheduling has a significant impact on the fair sharing of processing resources among virtual machines and on enforcing any applicable usage caps per virtual machine. This is specially important in commercial services like computing cloud services, where customers who pay for the same grade of service expect to receive the same access to resources and providers offer pricing models. However, the Xen hypervisor (and perhaps others) uses a scheduling mechanism which may fail to detect and account for CPU usage by poorly-behaved virtual machines, allowing malicious customers to obtain enhanced service at the expense of others.

We have demonstrated this vulnerability in the lab and on Amazon's Elastic Compute Cloud (EC2). Under laboratory conditions, we found that the applications exploiting this vulnerability are able to utilize up to 98% of a CPU core, regardless of competition from other virtual machines. Amazon EC2 uses a patched version of Xen, which prevents the capped amount of CPU resources of other VMs from being stolen. However, our attack scheme can steal idle CPU cycles to increase its share, and obtain up to 85% of CPU resources (as mentioned earlier, we have been in discussions with Amazon about the vulnerability reported in this chapter and our recommendations for fixes; they have since implemented a fix that we have tested and verified). We describe four approaches to eliminating this vulnerability, and demonstrate their effectiveness and negligible overhead. Finally, we give an algorithm in conjunction with our attack to discover the co-placement of VMs, this important mechanism can be utilized to conduct coordinated attacks in Cloud.

CHAPTER 3

DNSCloud

3.1 Introduction

The Domain Name System (DNS) resolves human-readable hostnames into machine-readable IP addresses as well as providing other information about domain names, such as mail services. As it behaves as a phone book for the Internet, the proper operation of DNS is fundamental to the maintenance and distribution of the addresses for the vast number of nodes around the globe. Thus, the DNS is a critical piece of the Internet infrastructure – nothing on the Internet would work if the DNS were to be targeted by attacks. There are several facets to DNS security. In this paper we focus on one of the most dangerous types of attack – DNS cache poisoning.

DNS cache poisoning refers to the type of attack where attackers inject false information into DNS cache, and results in inaccurate DNS lookups. Potential consequences of cache poisoning include identity theft, distribution of Malware/Spyware [37, 65], dissemination of false information and Man-in-the-middle attack [58].

In 1993, Christoph Schuba outlined several vulnerabilities including the technique of DNS cache poisoning in his thesis [116]. In the early version of DNS, it was possible to provide extra information in a reply packet that would be cached by the DNS server. This allows an attacker to inject fake information into DNS cache and perform other attacks in consequences.

In 1997 a serious BIND vulnerability was found [90]. As BIND did not randomize its transaction IDs, it is easy for an attacker to predict the next transaction ID after making their own request, and thus a cache poisoning attack could succeed by a spoofed query followed by a spoofed answer. To solve this problem, new versions of BIND use randomized transaction IDs.

In March of 2005 through the use of a cache poisoning attack people that were trying to access popular websites such as Google, EBay, CNN, and MSN were redirected to malicious sites that installed spyware on the victim computers [99].

Lately, Kaminsky attack [52, 74, 100] showed that DNS caches throughout the world were still susceptible to cache poisoning. This could lead to large-scale impersonation attacks by fake websites. Not only could it potentially lead to the personal and financial ruin of individuals but successful poisoning of this system could also result in catastrophic consequences at a national level [13, 52] were any infrastructural networks such as the military network domains to be targeted by an enemy.

Currently, there are two common ways to mitigate this vulnerability, patching the DNS server or patching the DNS protocol itself. Patching the DNS protocol, i.e. substituting DNSSEC [85] instead of DNS, requires a forklift upgrade at both the client and resolving name-server ends and for now the world has settled for updating and patching the DNS server software (typically BIND [85]). The problem with the patch however, is that it requires upgrades to all client side resolving name – servers, which is the more vulnerable and numerous end of things.

Hence it is clear that there is a need for a solution that does not require upgrading the software at the resolving name-server end. This will enable easy adoption of the solution. At the same time the solution must be robust enough to fend off large-scale attempts to poison the cache.

We propose the use of a distributed network, HARD-DNS, to dramatically reduce the probability of suffering from a poisoned cache in the DNS. We propose to leverage HARD-DNS machines as the public resolvers, by massively distributing the resolution functionality. In this fashion, the client's or ISP's name server will never be exposed to the attacker. When a user types in a domain such as www.foo.com into their browser, their name server contacts a random HARD-DNS machine which then recursively resolves the name on their behalf and returns the answer to the customer's name server (or resolver software).

Chapter outline: Section 3.2 briefly introduces DNS and cache poisoning. Section 3.3 discusses related work. Section 3.4 describes the details of DNSCloud. Section 3.5 presents experimental results in PlanetLab and we conclude in Section 3.6.

3.2 Background

3.2.1 DNS overview

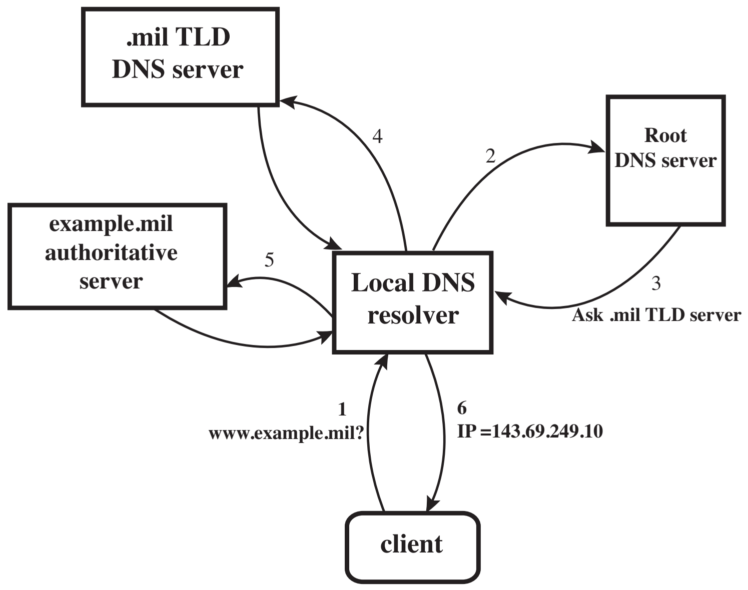
Although Internet and TCP/IP use IP addresses to route packets, locate and connect to hosts, users typically prefer to use friendly names. Obviously, remembering a single IP address such as 208.77.188.16 is not difficult, while as the number of hosts on Internet grows dramatically, tracking all IP addresses is not practical for end users. Instead, domain names, e.g. www.example.com, are used to refer to hosts which we want to connect. Thus we need a system to transfer user friendly domain names to IP addresses.

The Domain Name System (DNS) is a distributed hierarchical naming system for computers, services, or any Internet resource. Most importantly, it translates domain names meaningful to humans into the numerical (binary) identifiers associated with networking equipment for the purpose of locating and addressing these devices world-wide. An often used analogy to explain the Domain Name System is that it serves as the "phone book" for the Internet by translating human-friendly computer hostnames into IP addresses.

Figure 3.1 depicts a typical domain name/IP address resolution process.

- *step 1:* A client, attempts to visit a web page, for instance www.example.mil and needs to know the web pages IP address. The client (web browser) con-

Figure 3.1: Domain name resolution using DNS



tacts the DNS resolver belonging to its ISP (i.e., local DNS name server).

- *step 2:* If the local DNS server knows the location it supplies it and the request completes. If it does not have the answer in its cache, it will then contact one of the DNS root servers.

- *step 3:* Root servers are scattered in locations throughout the world serving the Internet. Each root server is responsible to serve a particular geographical location and is operated independently. The root server does not contain the IP address, but it knows who the authoritative DNS name server for the .mil zone, or top-level domain (TLD), are and it will send the resolver this information.

- *step 4, 5:* The resolver will then contact of the .mil TLD name servers which will reply with the authoritative name servers for the example.mil domain. The example.mil authoritative name server does know the IP address for www.example.mil and will send IP = 143.69.249.10 as the reply to the ISP name server.

- *step 6:* Finally, the ISP name server will send IP = 143.69.249.10 to the requesting client.

DNS lookup is time consuming and inefficient. To prevent unnecessary queries from being sent, the ISP name server can cache previously received replies and use those during the resolution process if necessary. For instance, if the client were to request the IP address for www2.example.mil, the resolver can now contact the example.mil authoritative DNS server directly. However, if the resolver caches a false IP address, then all the users in the same ISP querying that IP will be redirected to a false destination, and that is the goal of DNS cache poisoning attack.

3.2.2 DNS cache poisoning

A Cache Poisoning attack is an attack where the attacker fools the DNS server to cache a false IP address of a domain name. This causes all future requests to that DNS server for that resource to be sent to wherever the attacker chooses. An earlier survey [107] shows that a correct lookup of a domain name depends on a surprisingly numerous 46 servers on average. A few vulnerabilities have been found in the design of DNS that allow record injection without compromising the entire system.

- *Additional information:* attackers could attack malicious records as additional information to a query reply and some server implementations will cache it [116, 124].

- *Sequential transaction ID:* DNS uses the UPD transaction ID for reply authentication. Since earlier versions of DNS systems use sequential transaction ID, an attacker could guess the right transaction ID and send fake reply to the server [122]. This attack scheme has lower possibility to succeed in the system which uses random transaction ID or multiple queries.

- *Buffer overflow:* stack-based buffer overflow vulnerabilities of early versions BIND allow an unauthenticated attacker gain total control of the server.

In July 2008, at the Black Hat Conference held in Las Vegas, the Kaminsky DNS vulnerability was announced. This vulnerability allows an attacker to redirect network clients to alternate servers of his own choosing, presumably for ill ends.

The basic idea of the Kaminsky attack is to spoof answer packets from the authoritative name server to the requesting name server for a name in the authentic domain. The victim name server is provoked to go to a name server run by the attacker by either using it to recursively resolve a name in the attacker's domain or by putting a dummy pixel on a webpage belonging to the attacker's domain. When the victim server queries the attacker's name server the attacker learns the victim's IP address as well as source port. The attacker then provokes the name server to request a fake name in a real domain. Now the attacker sends a flood of spoofed response packets to the victim name server in an attempt to get a packet in before the genuine answer from the authoritative name server. The reason a flood of packets has to be sent is that the attacker has to get the TXID, a randomly chosen two-byte field, correct. If the attacker can get its packet in before the real packet then it can not only change existing cache entries corresponding to genuine names, it can even change the IP addresses for authoritative name servers for the domain. In this way all clients of the victim name server will end up going to the impostor web sites set up by the attacker. The Kaminsky attack can be used to hijack potentially even the root servers and therefore the entire root domain.

Figure 3.2: Flow of the Kaminsky Attack

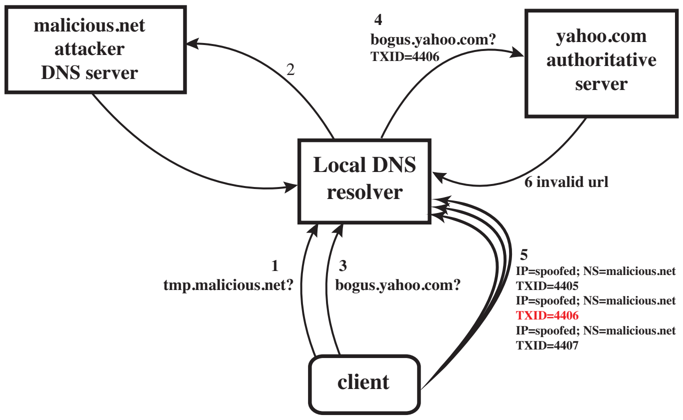

malicious.net attacker DNS server
2
4
bogus.yahoo.com?
TXID=4406
yahoo.com authoritative server
Local DNS resolver
6 invalid url
1
tmp.malicious.net?
3
bogus.yahoo.com?
5
IP=spoofed; NS=malicious.net
TXID=4405
IP=spoofed; NS=malicious.net
TXID=4406
IP=spoofed; NS=malicious.net
TXID=4407
client

Figure 3.2 shows a diagram of the Kaminsky attack.

- *step 1:* The attacker lures a client in the victim DNS server's domain to generate DNS lookup queries to resolve an url that is in the domain controlled by the attacker.
- *step 2:* As the local DNS does not know the domain name, it queries the attacker's name server and thus exposes its IP and source port.
- *step 3, 4, 5:* At the same time, the attacker floods the victim name server with forged packets that will answer its question to a real name server, each with a guessed TXID. If one of the forged packets contains the correct TXID (4406), then the victim DNS server will accept it and update its records. This forged packet not only contains IP address for the bogus host, but it will also contain information that points the real name server to a host controlled by the attacker.
- *step 6:* Finally the reply from the authoritative name server arrives too late, it will automatically be discarded as duplicates.

To increase the chances of this attack succeeding, a bandwidth attack, or Denial-of-Service (DoS) attack can be performed in parallel against the authoritative name servers. This will slow down the reply from the authoritative name servers to the victim name server by increasing the time window during which it can send the forged packets.

3.3 Related work

DNS [85] was not originally designed with security in mind, instead it was designed to be a scalable distributed system. To address this shortcoming, Domain Name System Security Extensions (DNSSEC) [43] was developed to provide security for certain kinds of DNS information, such as origin authentication and data integrity, but its implementation requires a forklift upgrade at both the client and resolving nameserver ends. DNSSEC is currently being deployed by the US Federal Government with the aim of having all Federal .gov domains and sub-domains configured to use DNSSEC by December 2009 [13][1]. Other agencies of the US Government, such as the Department of Defense (DoD) are also in the process of deploying DNSSEC, but certain types of networks (e.g. tactical networks), do not have the required infrastructure in place and require alternative means of protection against DNS cache-poisoning in the interim.

Unfortunately is not clear whether DNSSEC will ever fully supplant DNS. In the meantime the Internet continues to be susceptible to a variety of DDoS and cache poisoning attacks [81]. The Kaminsky attack, a cache poisoning attack, has been detailed extensively [52, 74, 100]. Later on a formal analysis of Kaminsky attack with probabilistic model checker was performed [98] performed .

Techniques for preventing DDoS [44] attacks using CDNs [10, 12, 81] and Cloud Infrastructures [1,7] have been studied but these are in the context of content delivery and not DNS.

A few schemes have been proposed to improve resilience in DNS. CoDNS [102] is based on Beehive, it masks delays in legacy DNS by diverting queries to other healthy resolvers. Schemes [39, 108, 128, 140] built on top of peer-to-peer networks have also been proposed to enhance fault-tolerance and load-balancing properties. The common of those schemes is to provide an alternative for the current DNS system. By exploiting the scalability and reliability of the underlying overlay network, those systems achieve faster lookups and fewer failures. However, non of those detects DNS cache poisoning attack. In the real world, replying on peers for DNS queries is not realistic and dangerous as it is difficult to find reliable and trustable peers. WSEC DNS [103] and [73] proposed schemes to protect recursive DNS resolvers from poisoning attack. WSEC DNS uses wildcard domain names, the basic idea is to prepend a random string to the queried domain names and still obtain a correct answer.

[1]By the time we submit this thesis, several federal agencies still have not secured their domains to protect users from domain name hijacking and cache poisoning attacks.

Game-theoretic approaches tailored for DNS-specific attacks also exist in the literature [63, 118, 123]. To the best of our knowledge, this work is the first to propose a solution to the cache poisoning problem by employing a separately hardened distributed network for name resolution in conjunction with the use of quorums and IP-cloaking.

3.4 System Design

The objective of the proposed Highly-Available Redundantly Distributed DNS (HARD-DNS) approach is to develop a robust protection technique that will minimize the chance of suffering from cache-poisoning attacks. HARD-DNS achieves this by:

- it requires no software upgrade or patch; with minimal configuration changes on the (client) side of the resolving name-server, it can easily be rolled out to the entire base of vulnerable DNS systems.

- it is a low-cost incremental approach that can be bootstrapped from an initial network of only a few machines. Of course, the more machines that are part of the HARD-DNS distributed network the more the robustness of the solution against DDoS attacks.

- it increases performance and reduces congestion on the Internet ultimately enhancing end-user experience.

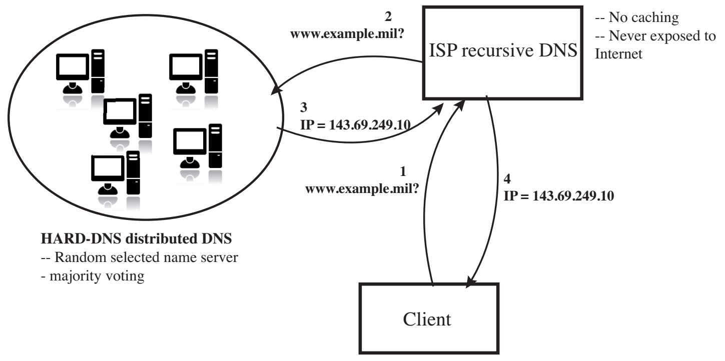
- it provides security both against known attacks as well as zero-day attacks since the task of resolving is essentially outsourced to the HARD-DNS network.

These benefits could come at the cost of a performance penalty since client side name-servers no longer cache resolutions. However, as we show in the experimental evaluation section 3.5, a well distributed HARD-DNS network incurs minimal overheads.

3.4.1 Overview

The basic idea is to solve the DDoS and cache-poisoning attacks by outsourcing the task of resolving name lookups to a fault-tolerant distributed network, HARD-DNS. We propose to leverage HARD-DNS machines as the public resolvers, by massively distributing the resolution functionality. This will require no software or hardware change whatsoever on the side of the resolving name servers. Name servers are configured with a hint zone consisting of the names and IP addresses of the root servers. We propose to replace this with the names and IP addresses of HARD-DNS nodes. Thus the name server contacts these CDN nodes to get its resolutions. The HARD-DNS nodes are set up to query recursively and then combine their answers using majority voting to diffuse the effect of poisoning. Observe that the original name server is completely shielded from the Internet because it never directly queries any other name server. HARD-DNS nodes resolve their queries by contacting the appropriate authoritative name servers. Figure 3.3 shows the overview of DNS lookup in HARD-DNS.

Figure 3.3: Flow of lookup in HARD-DNS



If the attacker were to attack or attempt to cache-poison they would instead attack a particular HARD-DNS machine. This would, at most, poison the cache of that particular HARD-DNS machine and since the customer picks a random HARD-DNS machine each time, they would be safe the overwhelming majority of the time. This will render the cache-poisoning attacks ineffective. The HARD-

DNS network uses overlay routing to distribute the answers amongst themselves and arrive at quorums and consensus. Overlay routing confers superior benefits of speed and reliability over native BGP routing. To compute its response HARD-DNS polls a subset of its name servers and then returns the majority vote. This would fail only if the attackers are able to poison the caches of all but a small fraction of the HARD-DNS machines, a virtual impossibility since HARD-DNS can be architected to have tens of thousands of machines. HARD-DNS also utilizes load-balancing techniques to diffuse the effect of DDoS attacks and IP-Cloaking techniques to hide the requests from the resolving name-server to the HARD-DNS network. IP-Cloaking provides greater security than port randomization which is the prevalent technique for mitigating the Kaminsky attack.

3.4.2 BGP Anycast and overlay routing

In our system, requests from the (client) resolving name server are directed to the optimal HARD-DNS server using BGP Anycast [23, 114]. The HARD-DNS machines selected for each request are different. The entire HARD-DNS network also functions as an overlay network and content is moved through the network from one end to another via intermediate relays. Even though each hop between CDN nodes is dictated by BGP nevertheless overlay routing is often superior to the native BGP routing between the same endpoints. This is because BGP is not aware of the congestion in the network.

3.4.3 Load-balancing and attack-diffusion

Load balancing among the HARD-DNS servers can be achieved using one or more layer 4 – 7 switches to share traffic among a number of DNS servers. This has the advantages of balancing load, increasing total capacity, improving scalability, and providing increased reliability by redistributing the load of a failed DNS server and providing server health checks.

The main advantage of a HARD-DNS network is that the entire process of resolution is offloaded from the (client) resolving name server thus shielding the location, even the existence, of the resolving (client) name server from the public Internet. Users and attackers only see the HARD-DNS nodes. Attackers can choose to attack HARD-DNS nodes but a well provisioned network will diffuse the attack through load balancing. When one of the nodes comes under attack HARD- DNS automatically does load balancing to move the load away to other servers. If the attacker chooses to move to the new servers then automatically the old servers spring back to life and start serving traffic. If the attackers stay with the original nodes then new traffic automatically gets served from other nodes. In this way, HARD-DNS protects the origin website against denial of service and degradation. The general mechanism of protection can be described by an analogy – just as important dignitaries are protected by big bodyguards that effectively act as bullet catchers so too does HARD-DNS protect the origin by absorbing the attack through its numerous and well-provisioned server clusters that distribute the load and diffuse the onslaught.

3.4.4 IP-cloaking

The common approach to dealing with the Kaminsky approach is to mitigate the chance of a spoofed packet having the right TXID by increasing the space of possibilities using port randomization. The resolving name server uses a random port from a space of a few thousand ports and since the spoofed packet has to get both the port and the TXID right to be able to poison the cache, this greatly increases the level of security. We extend this idea significantly using the concept of IP-cloaking. The HARD-DNS nodes are IPed with a large IP space. The resolving name-servers are then set up to employ hashing with a shared secret key to determine the specific IP address at the given time to contact. The HARD-DNS servers will answer only to the specific key at the specific time. The hashing can be done using a simple scripting language to modify the configuration file and does not require any software or hardware upgrade. The hint zone of root servers is periodically updated with the list derived from this shared key and soft-uploaded to the name server (name servers are already set up to soft-upload new configuration files) so that the original name servers contact the appropriate IP addresses. This provides an additional layer of protection because even if the IP addresses of the origin name servers are exposed nevertheless the attacker must correctly guess the HARD-DNS IP addresses derived from the shared key to be able to spoof their response – an impossible task for a sufficiently large IP space with which the HARD-DNS nodes are IPed (e.g. a Class B address space with 64000 IP addresses).

3.4.5 Quorum technique for fault-tolerance

We use a quorum based approach to reduce the chance of cache poisoning. For each resolution request from a (client) resolving name server, HARD-DNS utilizes a number of its servers, each querying for the resolution and then takes the majority response from them. Suppose we say that $2k$ HARD-DNS nodes query for the requested resolution and the majority response is returned back to the original name server. This response can be poisoned only if at least $k+1$ of the HARD-DNS nodes got poisoned. Studies of the Kaminsky attack show that it succeeds in about 10 seconds in the most optimistic case. If we assume that a server gets poisoned with $p = \frac{1}{10}$ in a given second then the probability that at least $k+1$ of them get poisoned is at most

$$\binom{k+1}{2k} \times \frac{1}{10}^{(k+1)} <= 10^{-(0.8k)}$$

3.4.6 HARD-DNS and CDNs

Observe that HARD-DNS can be easily built on existing distributed platforms such Content Delivery Networks (CDNs) or Cloud Infrastructures. CDNs have an edge over Cloud Infrastructures since they internally use DNS which can be easily modified to support HARD-DNS and they already posses a redundant and robust global footprint. It must be noted that we only rely on the CDN's DNS infrastructure and not on the CDN's content caching infrastructure. Furthermore, HARD-DNS truly requires only a large set of redundant DNS name servers; if this infrastructure can be provided by other means such as through a Cloud Infrastructure provider, a commercial CDN is not a necessary requirement.

3.5 Evaluation

3.5.1 Experimental setup

PlanetLab [22] is an open platform available to the academic community for distribution system studies, implementing and testing new large scale services. It runs over the common routes of the Internet and spans nodes across the world, make it more realistic than simulation. To evaluate HARD-DNS performance, we deployed it on PlanetLab. The advantage of using a platform such as PlanetLab as opposed to just testing in the lab is that we can evaluate HARD-DNS in the context of real world congestion and losses. We set up the experiment environment as follows :

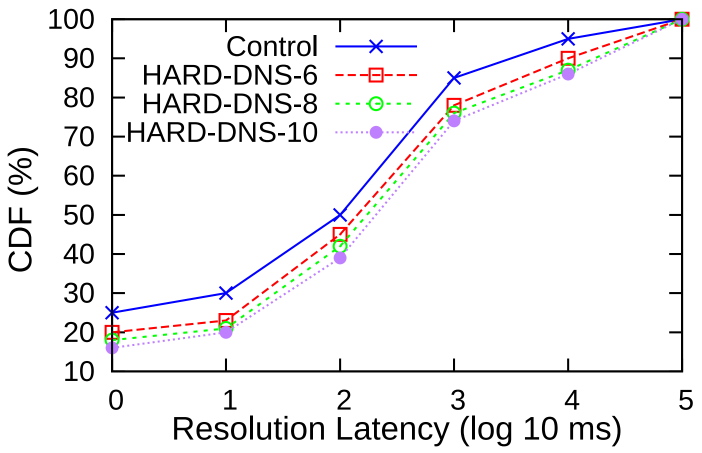
- We set up a HARD-DNS network of 6,8 and 10 servers on geographically distributed nodes.
- We used BIND version 8 which is a legacy version with the vulnerability to the Kaminsky attack.
- We used 1 server by itself as the control case.

For a network of $2k+1$ servers, we used a random subset of $k+1$ as the quorum, i.e., with 10 HARD-DNS servers the resolving name server would contact 6 randomly chosen machines.

We were continuously bombarding all the servers with the Kaminsky attack. The duration of a poisoning is a function of the TTLs (Time-To-Live). Since our goal is to see how quickly a given network (control server vs. HARD-DNS) gets poisoned but not how long it stays poisoned we would reset all machines every time we received a poisoned response from either the control server or the HARD-DNS network. We set long TTLs (10 seconds) on all returned responses which means that when a server got poisoned it would normally stay poisoned for a long time (10 seconds) but then we would reset all machines every time we detected a poisoned response, as explained before. We ran requests every 1 second for a resolution from both the control server as well as from HARD-DNS network, for a total of 12 hours. We measured the latency of lookups as also the time to poisoning. Our results should be viewed in the context of our setup on PlanetLab where we were averaging about 120ms RTTs (Round-Trip Times).

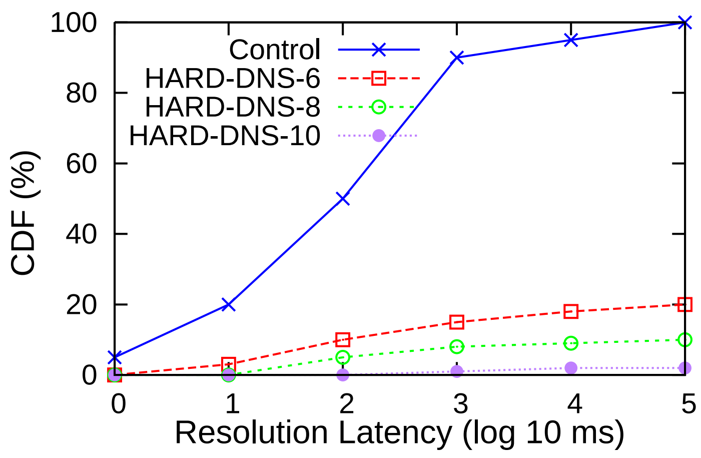
3.5.2 Results

Figure 3.4: Performance of HARD-DNS lookup



We now present results on the performance and reliability of the HARD-DNS network vs the control server. Figure 3.4 shows the cumulative distribution of lookup latencies incurred by HARD-DNS vs. the control server. As the number of machines in HARD-DNS system increases, the name resolution latency increases in a small range. Compared to resolution latency from the original single DNS server, where 80% of requests were processed in around 100ms and 100% of requests were complete within 10 seconds, a HARD-DNS system with 10 machines handles 80% of requests within 1 second, which is a little bit higher than control, but it is still able to process all the requests within 10 seconds.

Figure 3.5: Performance of HARD-DNS against Kaminsky



As both single DNS server and HARD-DNS system could handle 100% requests in 10 seconds, which means if any attack would successfully guess transaction ID and inject false information to DNS resolvers, it should be complete in 10 seconds, before the correct answer from authoritative name server comes back to DNS resolver. Figure 3.5 shows the cumulative distribution of the time to poisoning. With original single DNS server, Kaminsky attack could 100% succeed in 10 seconds, which means a large amount of queries result would be polluted. In contrast, HARD-DNS system efficiently protect itself from cache poisoning. By deploying HARD-DNS with 6 machines, at most 20% queries would receive fake replies. While with 10 machines, HARD-DNS shows high resilience against Kaminsky attack. Therefore, HARD-DNS provides reasonably low latencies while exhibiting great resilience.

3.6 Conclusion

The normal operation of the Domain Name System is vital for the Internet. Without DNS, the whole Internet would halt. We rely on DNS to provide correct domain name to IP mapping in daily life, so that we can surf on the Internet, browse web sites, exchange emails, etc. DNS queries starts from an end user (actually his browser) sending a request of a domain name to the local DNS server, then the local DNS obtains the IP address from DNS resolver and gets back to the user. As DNS query process is expensive, caching the query results efficiently saves Internet traffic and improves lookup performance. However, cache poisoning attacks could inject false information to DNS server and influence all the users sending request to that server.

The two common ways of solving cache poisoning problem is either patching DNS software or patching DNSSEC. DNSSEC is complex to implement, increases the size of DNS response packets and the answer validation increases the resolver's workload. The problem with the patching DNS however, is that it is difficult to upgrade resolving name-servers that are in critical operational paths and need to be running 24/7. While this patch reduces the risk of DNS poisoning by an attacker (by using randomization over a large port space), the patch by itself does not eliminate the threat from a formidable enemy who has the ability to bombard the resolving name-server with large quantities of spoofed responses.

We propose HARD-DNS, a distributed network to develop a robust protection technique that will minimize the chance of suffering from cache-poisoning attacks. The basic idea is to outsourcing the task of resolving name lookups to a fault-tolerant network, HARD-DNS can be implemented as a standalone network or built on top of platforms such as Content Delivery Networks (CDNs), or Cloud Infrastructures. We evaluate HARD-DNS performance against Kaminsky attack on PlanetLab, which gives realistic network environment setting. We proved that HARD-DNS efficiently prevent cache poisoning attacks as well as involves minor lookup performance overhead.

CHAPTER 4

WebCloud

4.1 Introduction

Over the past few years, we have witnessed the beginnings of a shift in the patterns of content creation and exchange over the web. Previously, web content—including web pages, images, audio, and video—was primarily created by a small set of entities and was delivered to a large audience of web users. However, recent trends such as the rise in popularity of online social networking; the ease of content creation using digital devices like smartphones, cameras, and camcorders; and the ubiquity of Internet access have democratized content creation. Now, individual Internet users are creating content that makes up a significant fraction of Internet traffic [17, 47]. This shift in content creation is leading to a change in the patterns of content exchange as well: users are much more interested in the content created or endorsed by others nearby in the social network. The result is that compared to content shared over the web just a few years ago, content today is generated by a large number of users located at the edge of the network, is of more uniform popularity, and exhibits a workload that is governed by the social network.

Unfortunately, existing content distribution architectures —built to serve more traditional workloads—are ill-suited for these new patterns of content creation and exchange. For example, web caches have been shown to exhibit poor performance on social networking content [36, 143], due to the more uniform popularity of content, causing many online social networking sites to begin to move away from content distribution networks (CDNs) and towards highly-engineered in-house delivery solutions [55, 75, 132].

In this chapter, we take a step towards addressing this situation by introducing WebCloud, a content distribution system designed to support the workloads present in existing online social networking websites. WebCloud works by recruiting users' web browsers to help serve content to other users. The key insight in WebCloud is to leverage the spatial and temporal locality of interest between social network users. Due to the geographic locality that often exists between friends in online social networks [83, 135], content exchange in WebCloud often stays within the user's local Internet Service Provider (ISP), thereby providing a bandwidth savings for both the site and the ISP. Moreover, WebCloud is naturally scalable, as each additional user provides additional resources. As a result, WebCloud provides most of the benefits of large centralized CDNs with lower costs.

WebCloud is built using two components. First, to enable browsers to help serve content, the provider includes WebCloud Javascript in their web pages. This Javascript caches content that users view. Second, to allow browser-to-browser communication, WebCloud uses a *redirector proxy* placed in the ISP, much in the same manner as existing CDN servers. However, in contrast to such servers, the proxy is not required to store any content, and is only present to enable communication between browsers. As WebCloud is built using standard web technologies, it is deployable today without browser changes or additional client-side software.

We evaluate WebCloud using four techniques. First, using microbenchmarks, we demonstrate that the latency of serving content via browsers and the storage present in WebCloud are sufficient to serve social networking content. Second, we simulate a large-scale deployment of WebCloud using traces from a real-world online social network. We demonstrate that WebCloud has the potential to serve over 40% of the content requests, even if a CDN is used in conjunction with WebCloud. Third, we demonstrate the practicality of WebCloud with a prototype deployment to real users on Facebook. Fourth, we demonstrate that WebCloud can be practically deployed on mobile devices by evaluating a proof-of-concept iOS app.

The rest of this paper is organized as follows. Section 4.2 explores the changing workloads and discusses the implications of these findings, motivating the design of WebCloud. Section 4.3 discusses related work. Section 4.4 details the design of WebCloud, and Section 4.5 provides evaluations and Section 4.6 concludes.

4.2 Background

We now take a closer look at the the trends of data exchange using real-world content-sharing data.

4.2.1 Data set

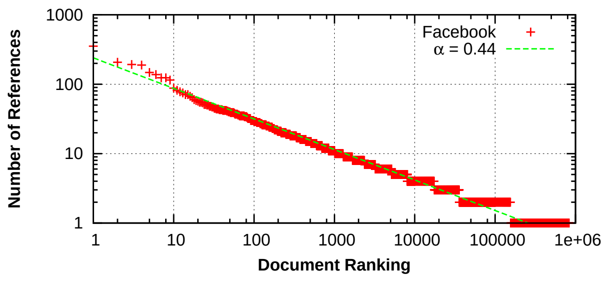
Our data set was collected on December 29, 2008 by crawling part of Facebook using the public web interface. We crawled the New Orleans Facebook regional network [93], which consists of users who claim to be located in the New Orleans area. Starting with a few randomly selected users, we conducted a breadth-first-search of all New Orleans network users, in the same manner as in previous work [92]. By default at the time of the crawl, Facebook allowed all users in the same network to view each others' profiles, and we were thus able to crawl a large portion of the New Orleans network. In total, we collected information on 63,731 visible users connected together by 1,545,686 undirected links.

As we are interested in content exchange, we also collected information about the photos that users exchange. Because data on photo *views* is not available, we use photo comments as a proxy for views (i.e., if a user has commented on a photo, they must have viewed it).[1] Crawling the news feed in a manner similar to previous work [135], we discovered information on a total of 1,068,787 comments placed on 816,508 different photos.

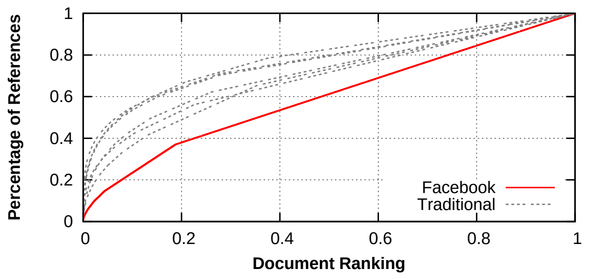
[1]Note that we are using photo comments (a visible form of interaction) as a proxy for measuring photo views (a latent form of interaction). Recent work [71] has shown that while visible interactions do not perfectly capture latent interactions, the two share a number of statistical properties and, in particular, are much more similar to each other than to the properties of the social network alone.

Figure 4.1: Photo popularity distribution on Facebook

(a) Distribution of photo popularity



(b) Cumulative distribution of document views



4.2.2 Properties

We now explore a few of the properties of this content exchange workload.

Content is created at the edge We first explore *where* the emerging content being exchanged over the web is being created. Today, the rapid adoption of smartphones, digital cameras, digital camcorders, and professional-quality music and video production software, combined with the low cost of broadband Internet service, has greatly eased content creation by individual users. Significantly more news articles are written by bloggers than news organizations [82], more photos are shared on online social networks [48] than on professional photography websites [14], and much of the content shared on YouTube, the most popular video-sharing site, is created by end users [28, 30] empowered by the ubiquity of webcams. The net result is that a significant fraction of Internet traffic contains content that is created at the edge of the network [17, 47].

Content is of more uniform popularity We now explore the *popularity distribution* of the content in emerging workloads, relative to previous workloads. To do so, we examine the popularity of photos on Facebook, and then compare the popularity distribution to that observed in studies of traditional web workloads. Figure 4.1a details the popularity distribution of the photos in our Facebook data set, and Figure 4.1b compares this to traditional web workloads [26]. We first note that, like traditional workloads, the Facebook request pattern follows a Zipf distribution, with an exponent of $\alpha = 0.44$.[2] However, there is one primary distinction with respect to traditional workloads: The Facebook workload contains a significantly lower exponent of the Zipf distribution (for reference, the exponents of the traditional web workloads range from 0.64 to 0.83 [26]). Thus, the newly emerging workloads have less emphasis on popular items, resulting in a more uniform popularity distribution and a significantly longer, fatter tail.

Exchange is governed by the social network We turn to explore *how* users are locating content. In particular, we explore the degree to which the exchange of content is governed by the structure of the social network. To do so, we calculate the fraction of comments on photos that come from the local social network of the uploader. The result of this analysis is that over 28.3% of the comments are placed by friends of the uploader, and at least[3] 89.1% are placed by friends or friends-of-friends (compared to expected values of 0.04% and 0.30%, respectively, were the placement random). This indicates that users are significantly more interested in the content that is uploaded by their friends and friends-of-friends. We have also found similar trends in browsing behavior in other online social networks: 80% of the photos viewed in Flickr were found by browsing the social network [92]. Furthermore, this analysis likely understates the impact of the social network, as we only consider photos that are publicly viewable; photos which are not publicly

[2]The exponent of the Zipf distribution was calculated using a maximum-likelihood estimator [34]. We observed a log-likelihood of -2602, strongly supporting a Zipf distribution.

[3]Our estimate of the fraction of comments placed by friends-of-friends is actually a lower bound: It is possible that two New Orleans users who are not friends or friends-of-friends within the New Orleans network are friends-of-friends when considering non-New Orleans users.

viewable are very likely to be restricted to only be visible to the uploader's friends, further increasing the fraction of local views.

Exchange has significant geographic locality Finally, we explore the connection between content exchange and *geographic locality*. Using our Facebook data set, we find that 32.9% of the friends of New Orleans users are also in the New Orleans network[4]; similar findings have been observed in other regional networks [135]. However, if we examine the fraction of content exchange that occurs between New Orleans network users, we observe that 51.3% of comments are placed by other users within the New Orleans regional network, even though only 32.9% of the friendship relationships lie within the network. This indicates that the significant geographic locality already present in social networks is present to an even greater degree in the content exchange that occurs over these networks. A similar conclusion was found in [136], by studying the traffic between Facebook U.S. datacenters and users, the authors found that traffic local to a region is produced and consumed significantly in the same region.

4.2.3 Discussion

The content that is increasingly being shared on the web today is created at the edge of the network, but is exchanged using centralized infrastructure. The usefulness of existing techniques on this workload is declining [36, 136, 143]: For example, caching the most popular 10% of the items in traditional workloads would satisfy between 55% [26] and 95% [19] of the requests; in our social network workload from the previous section, such a cache would only satisfy 27% of the requests. This also affects the ability to use CDNs, which similarly work best for popular content. As the amount and size of end-user-generated content increases, this centralized approach is likely to become a bottleneck, limiting the ability for users to exchange new and larger content.

[4]The actual fraction is likely even higher, as we only consider users who explicitly joined the New Orleans network.

Virtually all of the popular online social networks work in this manner: Today, every single piece of data shared on Facebook—every wall post, photo, status update, friend request, and video—is uploaded and likely served from one of Facebook's data centers [119]. It is no small irony that one of the world's most centralized architectures is being used to support one of the world's most naturally *decentralized* workloads.

However, we observed a strong correlation between the content views and the structure of the social network. This indicates that integrating the social network into the design of the content distribution system is likely to lead to techniques for better supporting the new content exchange workloads.

The most natural approach to address the changing workload is to work towards more decentralized content exchange. While some have suggested decentralizing the provider's data center architecture [136] into many regional data centers, this requires significant changes and expense for the provider. Instead, we focus on retaining the centralized provider architecture of today, while attempting to decentralize content exchange when possible. While a number of peer-to-peer content exchange applications exist (including decentralized social networks like PeerSoN [27] and Diaspora* [41]), their techniques are not applicable to web-based content exchange. Thus, in order to serve as a drop-in replacement for current distribution architectures, any new approach would need to work using web technologies. Unfortunately, web sites are designed around a client-server model, and web browsers are not designed to allow direct browser-to-browser exchanges.

4.3 Related work

4.3.1 CDNs

CDNs like Akamai, Limelight, Adero and Clearway offload work from the original website by delivering some or all of the content to end users, often using a large number of geographically distributed servers located in different ISPs. While most CDNs are operated in a centralized manner, systems such as Coral [49, 50] have been built which use decentralized architectures to accomplish the same task. These solutions are well-used, but they generally rely on resources donated by governments and universities, and are therefore not self-sustaining in the long run.

Other approaches have been explored allowing end users to participate in CDNs, including Akamai's NetSession, a client-side application that assists in content distribution to other Akamai clients, and FireCoral [127], a browser plug-in that participates in the Coral network. While the goals of both of these systems are similar to WebCloud, both require the user to download and install a separate application/ plug-in, limiting their applicability and userbase.

Similar to our work, Wittie *et al.* [136] proposed to reduce long latency and high loss on paths between users and the Facebook servers by serving content locally. Unlike our work, they did not utilize end users' capability but deployed local TCP proxies and cached content on those proxies.

Additionally, recent work [115] has demonstrated that information flow patterns over social networks (called *social cascades*) can be leveraged to improve CDN caching policies. This work is complementary to ours, and further demonstrates the importance of leveraging properties of the social network in future CDN designs. Finally, other recent work [104] has examined the benefits of allowing ISPs to assist CDNs in making content delivery decisions. This approach is similar in spirit to ours, but focuses on optimizing the server selection strategies employed by ISPs today.

4.3.2 P2P systems

WebCloud can be viewed as approximating peer-to-peer (p2p) content exchange though web browsers. Much previous work has focused on building standalone p2p systems for content storage and exchange [38, 84, 110] or avoiding the impact of flash crowds [121]. In those systems, clients generally have little choice of the peers where their data is stored, raising security, privacy, and reliability issues. To address these concerns, researchers have examined mechanisms for storing content between friends. For example, CrashPlan and Friendstore [129] provide cooperative backup systems that allow users store their data only on a subset of trusted peer nodes. However, unlike WebCloud, almost all work on p2p systems assumes a full networking stack and is therefore incompatible with being run inside a web browser.

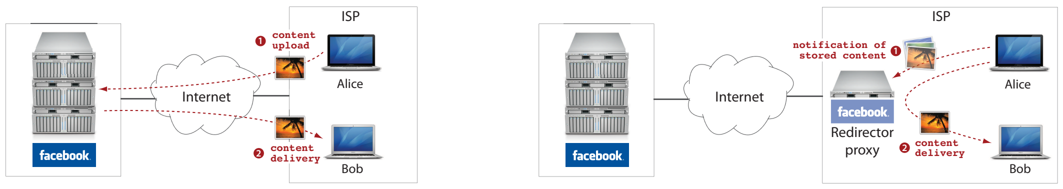
Additionally, researchers have exploited the use of social relationships in p2p systems for other applications, such as file sharing [105], email [53], DHT routing [86], and social networking [27, 41]. As people with close ties in social networks may share the same common interests, it is more likely for them to share interests or be mutually trusting. WebCloud leverages similar properties as these systems, but is designed for a different service.

However, the above studies concentrate on CDN performance in transmission of live streaming and large files, which does not suit the problem discussed in this chapter – people share small size photos. Also, in a CDN system, a proxy usually covers a large area and clients within that area are delivered the same content; but users of Facebook, Twitter, LinkedIn, etc., browse different content as they have different interests and friend groups. Clients in the same user group or community on social networks do not necessarily fall in the same geographical area, and user groups discussed in this chapter are relatively small – people only share their photos and information with their friends. Therefore, traditional CDN scalability and routing strategies do not resolve our topic.

4.4 System design

In this section, we describe the design of WebCloud, a system that takes the first steps towards decentralized content exchange on existing social networks.

Figure 4.2: Diagram comparing content sharing (a) in existing online social network, and (b) in WebCloud



(a) (b)

4.4.1 Overview

We begin by describing the deployment scenario that we expect for WebCloud. First, WebCloud is designed to be deployed by a web site, such as the provider of an online social network. We shall refer to the operator of this site as the *provider* for the remainder of this section. Second, we expect that the clients will access content using web browsers. In particular, we design WebCloud so that it is compatible with the web browsers of today. Third, WebCloud is designed to serve as a cache for content shared between users, much in the same manner as CDNs today serve as a cache for popular content. Thus, should WebCloud not be able to serve a particular request, we assume that the provider has a copy of the content located on their servers, and can serve the request. Given these assumptions, WebCloud serves as a drop-in component that is applicable to many large and popular providers, such as Facebook, MySpace, and Flickr.

4.4.2 Keeping content exchange local

The web operates in a client-server manner: without plug-ins, it is not possible to have one web browser fetch content directly from another. Thus, we first examine how closely we need to approximate direct communication between web browsers in order to address most of the concerns from the previous section.

Specifically, if we can keep the content exchange *completely within a single ISP's network*, we address many of the concerns in Section 4.2.3. In more detail, suppose that the two users who are exchanging content are served by the same ISP.

In this case, we argue that keeping the content exchange within the ISP, even if not directly between the users' browsers, is sufficient to reduce both the provider's and the ISP's costs. For the provider, keeping content exchange within the ISP obviates the need for the provider to serve the content, reducing the cost of bandwidth and serving infrastructure. For the ISP, keeping the content within its network removes the bandwidth cost of transit via another ISP. These observations are similar to those in the recent work on keeping peer-to-peer traffic local [137], as well as those used to place CDN servers.

4.4.3 Design

At a high level, WebCloud emulates direct browser-to-browser communication by introducing middleboxes called *redirector proxies*. These proxies—located within each geographic region of ISPs in the same manner as CDN servers—serve as a relay between web browsers, enabling communication and the transfer of content. Thus, instead of accessing content directly from the provider, the browser requests content from the closest redirector proxy. The proxy determines if any other online local user has the content, and if so, fetches the content from that user's browser and transmits it to the requestor. Should no local user have the content, the browser fetches the content directly from the provider.

Below, we describe the design of the redirector proxy, the browser–proxy message protocol, and the web site changes that are necessary to enable WebCloud. Throughout this section, we will refer to Figure 4.2, which gives an overview of the design of WebCloud.

4.4.3.1 WebCloud content

WebCloud is designed to be a drop-in component that can be deployed by existing web sites. WebCloud is agnostic to the type of content that is exchanged; however, WebCloud requires that content be named using content hashes (i.e., the name of a piece of content is its hash). This is necessary in order to ensure that content cannot be forged, and is discussed in more detail in Section 4.4.4.

4.4.3.2 WebCloud client

To deploy WebCloud, the provider includes the WebCloud Javascript library in their web pages. This Javascript serves two functions: communicating with the proxy, and storing and serving local content.

To communicate with the proxy, the WebCloud Javascript opens and maintains an active connection to the local redirector proxy.[5] This connection is based on XMLHTTPRequests (XHRs) using long polling. Using XHRs, the client always maintains a outstanding, unanswered XHR to the proxy (should this request time out, another is simply created). This allows the proxy to send messages to the client (by finally responding to this request with the content of the message), and the client to send messages to the proxy (by creating a new XHR with the message).

To store and serve local content, the WebCloud Javascript uses the LocalStorage API that is supported in most modern web browsers. In brief, LocalStorage allows a web site to store persistent data on the user's disk. LocalStorage is similar to cookie storage—in that it presents a key/value interface and is persistent across sessions—but is larger in size and can be programmatically accessed via Javascript more easily. When a user views content in WebCloud, the Javascript stores this content in the user's LocalStorage, treating it as a least-recently-used cache.

Finally, in order to take advantage of WebCloud, the provider must load content using WebCloud. To do so, the WebCloud Javascript code exports an API that the provider can use, shown below:

- **connect()** Called when the web page is first loaded. Causes the WebCloud library to connect to the nearest redirector proxy and establish an open session.

- **load(content_hash, id)** Called when the client code wishes to load an object.

[5]The local redirector proxy can be located using DNS techniques similar to those used by CDNs to locate the nearest CDN server.

Causes WebCloud to request the object from the proxy. If no peer has the content, the Javascript code loads content from the original web site. The *id* refers to the DOM id of the object; WebCloud will display the object once it has been loaded

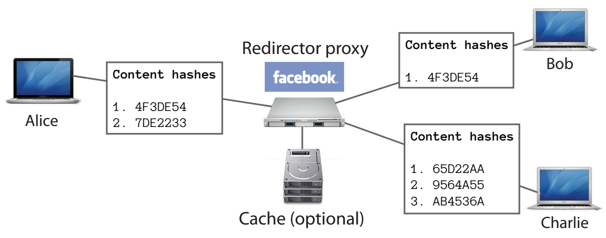
4.4.3.3 Redirector proxy

Redirector proxies allow clients to fetch content from each other. Each redirector proxy maintains connections to all of the provider's online clients that are located within the same geographic region and ISP.

The proxy keeps a list of the content that each client stores in LocalStorage, enabling the proxy to serve requests for content. A diagram of the state maintained by each proxy is shown in Figure 4.3. For example, in that diagram, if Alice requested content *AB4536A*, the proxy would fetch the content from Charlie and then deliver it to Alice. Should a proxy be unable to serve a request using the local clients (for example, if none of the connected clients were storing the requested piece of content) or its local cache, the proxy returns a *null* response to the client, who then fetches the content from the provider's servers in the same manner as today. Thus, the performance of WebCloud (measured by the fraction of requests served by a connected client) is heavily dependent on the browsing patterns of users and their online/offline behavior. We demonstrate in Section 4.5 that these patterns on today's sites are amenable to WebCloud, garnering a significant hit-rate.

The redirector proxy can also include an optional local cache, which is popu-

Figure 4.3: Diagram showing the state the redirector proxy keeps for each client, consisting of a list of the content ids each client is storing locally.



lated by the content it helps the clients exchange. Such a cache would be present if, for example, an existing CDN deployed WebCloud in conjunction with the already-deployed edge servers.

The domain name of the redirector proxy is chosen so that it is a member of the provider's domain. For example, a redirector proxy for Facebook would have a domain name in the *facebook.com* domain. Doing so enables the WebCloud Javascript to be able to communicate with the proxy (otherwise, Javascript's same-origin policy would prohibit communication).

4.4.3.4 Communication protocol

We now define the protocol between the WebCloud Javascript running at the client and the redirector proxy. The protocol consists of three message types, all encoded on the wire using JSON, described below:

- **update({content_hash, . . . })** The client sends an *update* message to the proxy to inform the proxy of the set of locally stored content. The message contains the list of content hashes that the client has in its LocalStorage, and the proxy records this list. The client can later send further *update* messages to the proxy, should the contents of its LocalStorage change (e.g., content is added or removed).

- **fetch(content_hash)** A *fetch* message can be sent either from the client to the proxy or vice versa, and contains the content hash of the content requested. When the client sends the fetch message to the proxy, the proxy checks to see if any other client has reported having that content in its LocalStorage. If so, the proxy forwards the fetch message to that client. If not, the proxy returns a *null* response to the requestor.

- **response(content_hash, content)** A *response* message is a reply to the *fetch* message, and contains the content hash and the content itself. This message can be sent either from the proxy to a client (who originally requested the content), or from a client to the proxy (in response to a forwarded *fetch* message).

A typical session would consist of a client connecting to the proxy and sending an *update* message to inform the proxy of the content that is locally stored. The client and proxy then exchange *fetch*, *response*, and *update* messages, as both the local user and other remote users browse content, and the client finally leaves by closing the connection.

4.4.4 Security

We now examine how WebCloud handles malicious users. There are three primary concerns: malicious users might attempt to serve bogus content to other users; malicious users may try to view content they are not allowed to; and malicious users might attempt to perform a denial-of-service attack by overloading the proxy or violating the WebCloud protocol. First, in order to detect bogus content, as discussed above, all content in WebCloud is identified by its content hash, so the proxy and all users are able to immediately detect and purge forged content. Second, using content hashes for naming also enables WebCloud to authenticate content requests, as users can only request content if they know its hash:[6] Given a sufficiently large hash space, it is extremely unlikely that a malicious user could “guess" an in-use content hash.

Third, in order to address users who attempt denial-of-service attacks by violating the protocol (e.g., by claiming to have content stored locally that they later turn out to not have), WebCloud uses similar techniques that are in-use by such sites today: Providers such as Facebook often block accounts, IP address, or subnets where malicious behavior is observed. Since the redirector proxy is under the control of the provider, existing defenses against these attacks can be deployed at the proxy as well.

[6]This is precisely the semantics that is followed by Facebook and other sites today; the URLs of images are obfuscated, but anyone who possesses the URL can download it.

4.4.5 Privacy

Next, we examine whether WebCloud changes the privacy implications of sharing content. First, WebCloud only allows users to fetch content that they could access anyway (see Section 4.4.4), so users cannot abuse WebCloud to view content they otherwise could not. As a result, WebCloud does not allow users to disclose content to unauthorized third parties who they could not already.

Second, in WebCloud, users do receive some information about *views* of content by other users (i.e., when a user receives a *fetch* message for a piece of stored content, he knows that some other user is viewing that content). However, due to the indirect nature of the communication, users are unable to determine *who* is viewing the content. Thus, WebCloud effectively provides k-anonymity [112] to browsing users, where k is the number of online users connected to the same proxy who are able to view the photo. Many providers already provide some form of view information to users (e.g., view counts, or names left beside comments on content), so WebCloud often does not leak information that was not already available.

Regardless, there may be corner cases where k-anonymity is insufficient (e.g., if a user only makes a piece of content visible to a single other user, effectively removing viewer anonymity). In such cases, the provider can disable loading such content via WebCloud (or can allow the browsing user to do so). Alternatively, the provider can configure the proxies to add background requests to random pieces of content, sometimes referred to as *cover traffic* [51], in order to obfuscate requests.

4.4.6 Mobile users

The description of WebCloud so far has focused on users who are connected from traditional web browsers. However, users are increasingly accessing services like online social networks from mobile devices. For example, dedicated apps exist for Facebook and MySpace on the iOS and Android platforms, enabling users to create and exchange content using their mobile devices. Below, we first address the technical issues associated with deploying WebCloud on mobile devices, then evaluate potential usage of power and bandwidth with a prototype of iOS application.

As the mobile web browsers on both platforms support both LocalStorage and XHRs, WebCloud works unmodified these devices when the user visits the provider's site. However, a potential drawback is that the session times are likely to be much shorter, as smartphones typically only allow users to have one “active" web page at a time (thus, whenever the user browses away from the site, closes the browser, or puts the phone to sleep, the connection to the redirector proxy will be broken). Luckily, the popularity of site-specific apps (e.g., Facebook for iOS) enables an alternative: WebCloud can be integrated into the app, and use the background service support provided by both platforms to maintain a connection to the redirector proxy.

4.5 Evaluation

4.5.1 WebCloud implementation

We implemented WebCloud to work in conjunction with Facebook's photo-sharing service. Photos are one of the most popular content-exchange mechanisms on Facebook, allowing us to easily obtain a userbase and a workload. The prototype redirector proxy consists of 1,283 lines of Python, most of which is the low-level communication support code for WebSockets and XHRs. The client-side support for WebCloud consists of 1,226 lines of Javascript. In total, the prototype of WebCloud took one graduate student four weeks to design, implement, and test.

In order to deploy our WebCloud prototype, we set up a web proxy that was configured to inject the WebCloud Javascript when serving Facebook's Javascript files. Thus, users installed WebCloud by configuring their browser to fetch content via this web proxy. Additionally, the web proxy ensured that all content requests for Facebook photos were served by WebCloud, thereby allowing WebCloud to naturally work with optimizations like photo pre-fetching. As a result, from the perspective of the browser, it appeared as if Facebook had deployed WebCloud.

Table 4.1: Average time to load Facebook photos for various configurations. The columns represent where the content is loaded from: Facebook as normal, a WebCloud (WC) client located on the same LAN as the proxy (WC-LAN), and a WebCloud client connected via a cable modem (WC-Cable). The rows represent where the content is loaded to: a computer on the LAN, and a computer connected via a cable modem.

Accessed from	Served from		
	Facebook	**WC-LAN**	**WC-Cable**
LAN	668 ms	63 ms	398 ms
Cable	690 ms	153 ms	532 ms

4.5.2 Microbenchmarks

4.5.2.1 Content loading latency

The first WebCloud microbenchmark that we examine is the latency incurred in downloading photos. Since loading photos using WebCloud requires the request to be routed via a redirector proxy and served by the browser of a remote client, we explore whether any additional latency is incurred. To do so, we uploaded a set of 10 new Facebook photos (average size 62 KB) and then downloaded them in one of three ways: from Facebook as normal, from a WebCloud client connected to the same LAN as the redirector proxy, and from a WebCloud client connected via a cable modem. We ran each of the above three tests twice, once downloading the photo to a machine on the same LAN as the redirector proxy, and once downloading the photo to a machine using a home cable modem connection. Note that we uploaded new photos for each experiment, in order to cancel out the any effects from caching. For each configuration, we report the average download time.

The results of this experiment are presented in Table 4.1. We observe a number of interesting trends. First, in all cases, loading the content using WebCloud was actually *faster* than loading it directly from Facebook. This results from running our experiment all on machines in Boston; loading data from Facebook requires downloading it from California. Note, though, that we expect one redirector proxy to be placed in each ISP region, so this is representative of what would be expected in practice. Second, accessing content that is stored on a client connected via a typical home cable modem added approximately 350 ms of additional latency in both tests, due to the slower connection. However, in all cases, the latency of WebCloud is acceptable.

4.5.2.2 Storage size

The second microbenchmark that we examine is the number of photos that can be stored in each user's LocalStorage. To do so, we first collected a sample of 954 Facebook photos, and found that the sizes ranged from 12 KB to 226 KB, with a median of 67 KB.[7] By default, Chrome, Safari, and Firefox all set a maximum size of 5 MB for the LocalStorage per (domain name, port) pair. Taking into account the 33% overhead induced by storing photos in base64 format, WebCloud is able to store 56 photos, on average, in each user's LocalStorage.

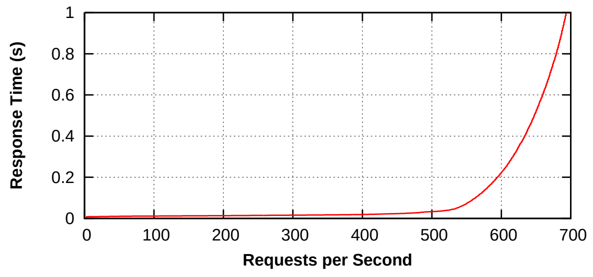
As the 5 MB storage limit is per domain and port, WebCloud extends the storage limit arbitrarily by loading stub Javascript from multiple domain names (e.g., *foo1.facebook.com*, *foo2.facebook.com*, ...) and ports, all pointing to the same redirector proxy. By setting the Javascript *document.domain* property appropriately, WebCloud ensures that these stub scripts can communicate, allowing the multiple LocalStorage instances to all be accessible. Regardless, as we demonstrate below, even using only 5 MB allows most of the savings of WebCloud to be realized.

4.5.2.3 Redirector proxy scalability

The third microbenchmark that we explore is the scalability of the redirector proxy. We are primarily interested in understanding the rate of *fetch* requests that a single proxy can handle, as this serves as the dominating factor controlling the number of online users that the proxy can support. For this experiment, we ran our redirector proxy implementation on an 4-core 2.83 GHz machine with 16 GB of memory. We

[7] Photos uploaded to Facebook are automatically resampled, resulting in a more uniform size distribution even though the source images are likely to vary greatly in size.

Figure 4.4: Average response time versus request rate for a single redirector proxy.



then connected an increasing number of clients to the proxy, all located on the same LAN, where each client was configured to continually issue content *fetch* requests for a 85 KB photo every five seconds. Finally, we examined the tradeoff between the rate of incoming requests and the response time (measured as the time elapsed from the start of the request until the last byte of the content arrives) at the clients.

The results of this experiment are presented in Figure 4.4. Our prototype redirector proxy implementation is able to support over 500 content *fetch* requests per second before significant additional latency is incurred. If we assume that the average user issues one content request per minute, this represents a single redirector proxy supporting over 30,000 online users. It is also worth noting that our proxy implementation is not optimized; a highly-optimized implementation is likely to support an much greater number of users.

4.5.3 Simulation

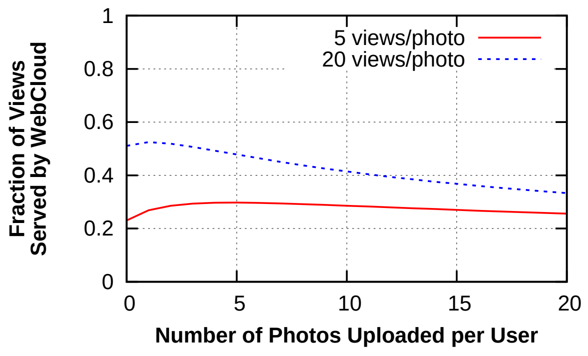
Next, we evaluate the potential of WebCloud at scale by simulating Facebook deploying WebCloud in one region.

4.5.3.1 Generating traces

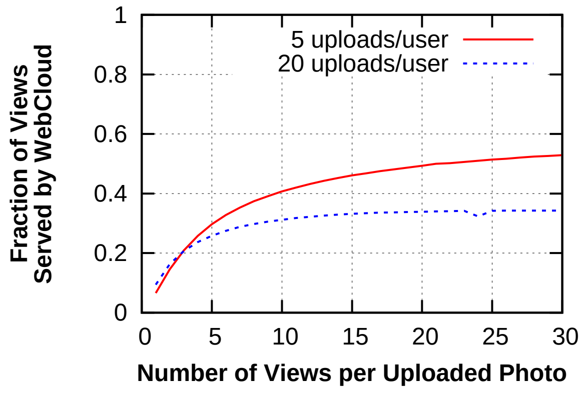
For the simulation, we need a number of pieces of data: a social network, a trace of when users are online and offline, and trace of when users browse each other's photos. Unfortunately, a detailed trace of Facebook user online/offline and photo viewing behavior is not widely available. Instead, we use our Facebook data dis-

Figure 4.5: Fraction of views served by WebCloud in various simulations.

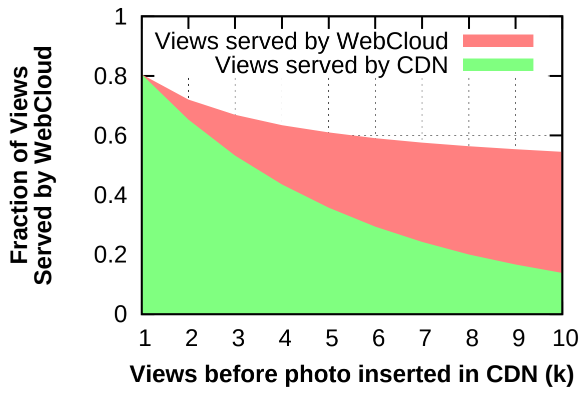
(a) WebCloud hit-rate as the average number of photos uploaded per user is varied



(b) WebCloud hit-rate as the average number of views per photo is varied



(c) WebCloud hit-rate when run in conjunction with a CDN, varying the minimum number of hits(k) before a photo is placed in the CDN



cussed in Section 4.2.1 to generate synthetic traces.

We simulate Facebook deploying WebCloud to 63,731 users in the New Orleans regional network. Each trace represents one week of activity within the New Orleans regional networks. The trace is generated so as to preserve the bias that is present in user online/offline behavior, photo uploads per user, and photo views per user. Below we detail our methodology for generating one-week synthetic traces in the WebCloud evaluation, and compare the traces to studies of real-world systems.

Photo uploads To generate a photo upload event, we need to select two things: the uploading user and the time of the upload. To select the uploading user, we choose randomly from the list of uploaders observed in the photo comments data. Users who were observed to upload more photos are more likely to be chosen (e.g., a user who uploaded five photos is five times more likely to be chosen than a user who uploaded a single photo). This method preserves the non-uniform distribution of photo uploads across users. Then, we randomly select an upload time during the simulated week, as we do not have the upload time of the photos.

Photo views To generate a photo view event, we need to select three things: the viewing user, the time of the view, and the photo that is being viewed. First, to select the viewing user, we use a similar mechanism as above, selecting randomly from the list of users who placed comments (again, this preserves the fact that certain users view photos more than others). Second, to select the time of the view, we pick a day of the week and time of the day randomly from the list of timestamps of the viewing user's comments. We add a small amount of random time (between -30 and 30 minutes) to this timestamp in order to ensure that multiple views do not happen at once. Selecting the time in this way preserves the daily and weekly trends in browsing behavior.

Third, to select the photo that the user views, we first select a *uploading user*, one of whose photos the viewing user will view. The uploading user is selected randomly from the list of users whose photos the viewing user commented on. This method preserves the fact that users are more likely to view certain other users' photos. We then select which of the uploading user's photos is viewed using a weighted distribution (since more recent photos are more likely to be viewed). The weights are derived from a study [91] of the views received by Flickr photos. In brief, photos were observed to receive 37.2% of their views on the first day, 21.3% on the second day, and so forth. Thus, all photos uploaded by the uploading user in the previous day have an even likelihood of being chosen, which is higher than all photos uploaded two days ago, etc.

Online/offline trace Finally, we generate an online/offline trace for the users based on the photo view trace just described. For each user, we look at the timestamps of their photo views. For each timestamp T, we simulate the user coming online at time $T - v_-$ and simulate the user going offline at time $T + v_+$, with v_- and v_+ selected randomly between 1 and 30 minutes. The result is that, for each view, the user is online for between 2 and 60 minutes (with an average of 30 minutes that approximates recent session time studies [66]).

Evaluating traces We now briefly compare the synthetically generated trace to empirically gather data regarding use of Facebook and other social media sites. First, we examine the average time spent per user online during the week. In all traces, the average time online ranged from 1.77 to 6.02 hours, which matches favorably with studies of real-world Facebook usage [72]. Second, we examine the number of online users during the course of our simulated week and observe the strong diurnal patterns that would be expected in realistic workloads. Third, we examine the popularity of individual photos themselves. The overall photo popularity distribution closely matches previously observed distributions of photo popularity in social networks [29].

4.5.3.2 Simulation results

There are two primary questions that we are interested in. First, what is the hit-rate of WebCloud? In other words, what fraction of the photo views can be served from another online user via WebCloud, instead of directly from the provider? Second, how does WebCloud compare to a CDN, such as an Akamai? In other words, if such a CDN is used in conjunction with WebCloud, what additional hit rate does WebCloud provide?

For these experiments, we only consider views that are not served from the user's local cache (i.e., if a user views a photo that is already present in his LocalStorage, that photo view is disregarded). We simulate all clients as being able to store 50 photos in their LocalStorage. To explore the various environments that WebCloud may be deployed in, we vary two parameters: the number of photo uploads and the number photo views.

For clarity, we express the parameters in terms of the number of users or photos in the network (i.e., the average number of photos uploaded *per user*, and the average number of views *per photo*). However, the uploads and views are generated using the process described in the Appendix, meaning certain users upload many more photos than others, and certain photos are viewed many more times than others. Finally, for all experiments in this section, we repeat the experiment 10 times with different random seeds and report the average.

Varying the number of uploaded photos We begin by examining how the number of photos that are uploaded affects the hit rate of WebCloud. To do so, we run simulations with two different settings of the average number of views per photo (5 and 20). As the average number of photos uploaded per user increases, we expect that the WebCloud hit-rate will initially increase as the collective LocalStorage fills. However, the hit-rate will then begin to drop off, once each user's LocalStorage fills up.

The result of this experiment is presented in Figure 4.5a. We see that for each setting, the hit-rate has a maximal peak, matching our intuition from above. However, we note two trends: First, the fall-off in hit-rate is rather slow, falling only by about 20% as the number of photos uploaded increases from 5 per user to 20 per user. Second, the overall hit-rate is surprisingly high, ranging between 23% and 57%. The upshot is that even in this wide variety of configurations, WebCloud serves a significant fraction of the views.

Varying the number of views per photo Next, we turn to evaluate how the average number of views per photo affects WebCloud's performance. As the the number of views increases, we expect that the hit-rate of WebCloud to increase, due to the bias towards viewing recently uploaded photos. Similar to the previous experiment, we present results from two different configurations of the average number of uploads per user (5 and 20). The result of this experiment is presented in Figure 4.5b. We observe that our expectation holds: as the average number of views per photo increases, the hit rate of WebCloud rises.

Comparison to a CDN To answer our final question, we modified our simulator to simulate a CDN, such as an Akamai edge node, running conjunction with WebCloud. In these simulations, clients first query the CDN; only if the CDN does not have to content is the request forwarded to WebCloud. However, it is unrealistic to assume that *every* photo would be inserted into the CDN, as sites today only use such services to serve popular content. Thus, we configured the CDN to only store content that been requested k times. Increasing k implies that only more popular content is served via the CDN. We assume that the CDN has unlimited storage, but an empty cache at the beginning of the experiment.

The results of this experiment are shown in Figure 4.5c, showing the fraction of the views served by the CDN and by WebCloud (with an average of 5 photos per user, and an average of 20 views per photo). We observe that if $k = 1$, the CDN caches and serves all content, forwarding no requests to WebCloud. However, if k is increased to 5, WebCloud serves over 25% of the requests, and at $k = 10$, WebCloud serves over 40%. This result shows that even if the provider uses a CDN, WebCloud still provides a significant hit rate. The take-away from these experiments is that WebCloud holds the potential to provide a significant benefit to online social networking website providers. For example, if the average LocalStorage is 5 MB, the average user uploads 5 photos per week, and the average photo is viewed 11 times in the first week (as we observe in random Flickr photos [91]), then we would expect that WebCloud would serve 43% of the requests. Moreover, even if we assume that WebCloud is deployed in conjunction with a CDN that caches photos that have been viewed at least 10 times, we would still expect that WebCloud would serve 40% of the requests.

4.5.4 Mobile devices

We now turn to examine WebCloud on mobile devices. Recall from our discussion in Section 4.4.6 that we are concerned with two questions: First, can WebCloud be feasibly implemented as a background service? Second, if so, what is the impact on battery life and data usage? To evaluate this, we implemented a prototype WebCloud app on iOS. Our app registers itself as a background VoIP service, allowing it to maintain a persistent connection with the redirector proxy even if the app is not in the “active" application.

To evaluate the impact on battery life, we deployed our WebCloud app to an iPhone 4 running iOS 4.2 configured to connect to a test redirector proxy. The proxy issued *fetch* requests for the app to serve a 60 KB photo every five seconds, allowing us to measure the number of photo requests that could be served from the phone before running out of battery. We found that the WebCloud app could serve 5,031 requests over 8.26 hours when connected via 3G, and 24,700 requests over 34.9 hours when connected via WiFi. Given the fact that, even in the most demanding scenario above, the user with the *heaviest* workload only served 160 photos per day (and the average user served 4.36 photos per day), the WebCloud app is likely to only consume a small amount of the battery life. Finally, the data from our simulations also addresses the concerns over data usage. Even under the heaviest workload, the most-loaded user in our simulations served a total of 72 MB during the simulated week, while the average user served 2 MB. Both are within the data allocation provided by most 3G providers.

4.5.5 Deployment

As a final point of evaluation, we deployed our WebCloud prototype on a small scale within our department at Northeastern University to examine how it would work with real-world users.[8] We first set up a redirector proxy within the department, and then recruited users by emailing our graduate student populations. In total, over the course of our 10-day deployment, we observed 17 users install WebCloud on a range of browsers and operating systems. These users connected a total of 585 times to the proxy, and browsed a total of 2,069 photos with an average session time of 18 minutes. 539 (or 26%) of these photos were served from another WebCloud client; the remaining 74% of the photo views were fetched from Facebook directly. While this fraction of WebCloud is lower than in our simulations, it is likely due to our deployment environment: For the simulation, we considered what would happen if Facebook deployed WebCloud to an entire region; in our real-world deployment, we had to manually recruit people, so our social network coverage is substantially lower. However, the deployment demonstrates that WebCloud is feasible with today's OSN sites and web browsers.

4.6 Conclusion

In this thesis, we took a step towards decentralized web-based content exchange by introducing WebCloud. WebCloud makes novel use of established web technologies to allow clients to help serve content to other clients, keeping the content exchange local and providing a savings for both the site provider and the ISP. The result is that WebCloud is able to serve as a drop-in component on sites like Facebook, Flickr, and MySpace. Microbenchmarks, simulations, and a small-scale deployment on Facebook demonstrated that WebCloud works well in practice, and that WebCloud holds the potential to serve over 40% of the views, even when used in conjunction with a CDN.

[8] Our real-world deployment was covered under Northeastern Institutional Review Board application #10-07-23.

However, smartphones present two unique challenges: Limited battery life, and data access charges. The background service support present on many mobile operating systems holds the potential to help with the former, as the app itself is not required to be running constantly, enabling the operating system to conserve battery life when no requests are received. Regardless, the impact of WebCloud on battery life is still unclear. In order to gauge this impact, we implemented a prototype of WebCloud on iOS. Fully described in Section 4.5.4, we demonstrate that the impact on battery life and data usage is acceptable.

4.6.1 Discussion

Caching at the proxy One potential question about WebCloud's design is *Why not simply cache everything at the redirector proxy, instead of in the users' LocalStorage?* Such a design would be simpler, and would approximate the design of the centralized CDNs of today. However, such a design has two downsides. First, as we observed in Section 4.2, caching in the network is much less effective for emerging workloads. Second, as users begin to share larger pieces of content (e.g., videos), such a cache would require significantly more storage and would quickly increase in cost. Using WebCloud, providers can push that storage cost onto the users themselves (who, in aggregate, have significantly more storage resources than any one proxy), resulting a more scalable system with lower costs. Moreover, we demonstrate in Section 4.5 that even if WebCloud is used in conjunction with a CDN, WebCloud still serves a significant fraction of the requests.

Number of online users One concern with WebCloud is the session time of users; should the average session time be short, it is unlikely that WebCloud would be able to serve many content requests since a large number of users may not be online at the same time. However, many social networks exhibit long session times (e.g., Facebook has been observed to have average session times of almost 30 minutes [66]). Additionally, these sites often provide incentives for users to stay online, such as the presence of instant-messaging facilities. The result is that on the sites that WebCloud is targeting, it is likely that users are online sufficiently often.

Deploying proxies Another question in WebCloud arises over who will deploy the redirector proxies placed in each ISP. Clearly, large providers like Facebook could deploy their own proxies if they choose, but this option is only practical for the largest providers. However, the redirector proxy could also be a service provided by existing CDNs like Akamai, as these CDNs already have the necessary servers deployed in numerous ISPs.

CHAPTER 5

SamaritanCloud

5.1 Introduction

The ubiquity of the Internet has revolutionized how people interact with each other. Since the early 70s, when the first email was exchanged between two computers sitting right next to each other, email has gradually become indispensable in our daily life. Since the spread of Bulletin Board Systems (BBS) and web browsers, people have started to share information and opinions online, *even* with others they may not know in the real world. From the days of the web's earliest social networking sites such as Geocities [138] to today's Facebook [46], Google+ [56] and Twitter [131], the Internet has always been about reaching people worldwide. Social networking sites provide the platform for serving a basic human need – the need to build social relations with people with similar interests and activities.

Social network sites have attracted millions of users; many of whom have integrated these sites into their daily habits. There is no denying the fact that social networking sites provide a variety of services that allow users to share ideas, activities, events and interests within their individual networks. They also enable individuals to interact with strangers in a relatively secure way. With the maturation of online social networks (OSNs), people have begun to form online communities and look for help through social networks from people who they do not *even* know in person.

The most common method to seek for help on social network sites is the *post* – an user posts his question or request on his social network page or a group page and waits for a response, which is very similar to subscribing to a newsgroup or broadcasting on an email list. Though the act of posting is simple enough nevertheless it has some disadvantages.

- Uncertain response latency: since users of popular social network sites, such as Facebook, live in different locations (worldwide) and have different schedules, a post on a group page may not reach all the members in a reasonable amount of time and further, may be overwhelmed by other, more recent, posts. Therefore, the waiting time is uncertain.

- Limitations on requests: as groups on social network sites are usually formed based on interest, occupation and gender, most online requests focus on recommendations and suggestions. It is hard to get responses for time-restricted and location sensitive questions. People do not make offer to help in the offline world until they trust one another substantially.

- Privacy loss: users expose their requests to a large group, including friends they know in real life and other users who may not even be in a position to offer help. Unnecessary privacy leaks that may affect user's personal life should be avoided.

Thus, a system that would be useful for people to help each other (especially physically) efficiently and privately in the real world is needed. Consider the following cases: When you see a parking ticket because you were 5 minutes late, do you wish someone could put in a crucial quarter for you? When you are locked out of the laboratory, do you wish you could know where your lab-mates are before calling them one by one for help?

We propose SamaritanCloud, a location-based cyber-physical network helping people to reach friends or strangers in desired locations for help. Unlike standard geosocial network services [3–5, 9] which focus on sharing or tagging in a specific environment, SamaritanCloud offers users remote hands and eyes access to somewhere where they are not. SamaritanCloud is deployed as a location-based cell phone application, the server coordinates users requests and efficiently find candidates who are able to offer help. With locality sensitive hashing and the encryption scheme we discuss in Section 5.4.3, SamaritanCloud preserves efficiency and privacy.

Chapter outline: Section 5.2 briefly introduces location based service and locality sensitive hashing algorithm. Section 5.3 discusses related work. Section 5.4 describes the details of SamaritanCloud. Section 5.5 presents experimental evaluations. Section 5.6 contains a discussion of the range of dimensions where locality sensitive hashing becomes relevant and we conclude in Section 5.7.

5.2 Background

5.2.1 Location based service

A location-based service (LBS) is an information or entertainment application that is dependent on a certain location. It is accessible with mobile devices through the network and can be used in a variety of areas such as research, entertainment and personal life. The main purpose of a location-based application is to provide service based on the geographic location of the user. These services offer users the possibility to find other people, machines and location-sensitive resources. A location request can be originated by the client himself or another application provider of the network operator. Usually the customer has to give permission for the location request to be satisfied.

Depending on the service, LBS can be user-requested or triggered. In a user-requested scenario, the user retrieves the position once and uses it on subsequent requests, e.g., map navigation. LBS can also be triggered automatically when the client is in a specific location, e.g. location may be updated when the client passes across the boundaries of the cells in a mobile network.

Depending on how the location information is used, three types of LBS can be distinguished: position-aware, sporadic queries and location-tracking. Position-aware services are based on the device's own knowledge of its position – these LBS applications are self-contained – a user device obtains a position and then performs some processing and presents the resulting data back to the user. Sporadic queries apply to services in which an individual initiates the transfer of position information to a service provider. These queries contain only the current position. Location-tracking applications will require the position to be continuously sent to a server either for display to other parties, processing or obtaining relevant content.

5.2.2 Nearest neighbor searching

The nearest neighbor problem is of major importance in location based applications, usually involving similarity searching. Typically, the features of object of interest are represented as a point in $\mathbb{R}^d$ and a distance metric is used to measure the similarities of objects. Given a set of n points $P = \{p_1, p_2, ..., p_n\}$ in a metric space X with distance function d, preprocess P so as to efficiently answer queries for finding the points within a range r from the query-point $q \in X$. The problem is more interesting in d-dimensional euclidean space where $X = \mathbb{R}^d$ under some l_s norm.

Locality sensitive hashing (LSH) provides efficient nearest neighbor search algorithms – finding similar entries in a large collection of data. The basic idea to is hash all the items so that similar items have higher possibility to be mapped to the same bucket. This approach belongs to an interesting class of algorithms known as randomized algorithms. So, like any other randomized algorithm, LSH algorithms do not guarantee the correct answer but instead provide a high probability guarantee that they will return the correct answer or one close to it.

A variety of data structures for the nearest neighbors were discussed in [113], including variant of k-d tree, R-trees and structures based on space-filling curves. While some algorithms perform well in 2-3 dimensions, despite decades of effort, the best deterministic solutions suffer from either space or query time that is exponential in d [16, 87, 89, 139]. When d is large enough, the algorithms often provide little improvement over O(n) complexity that compares a query to each point from the database. This is called "the curse of dimensionality".

To overcome this bottleneck, randomized LSH-based algorithms were proposed to solve approximate nearest neighbor searching [69, 78, 80]. The general form of the problem is that given a set of n points C, preprocess C such that given any query point $q \in C$, the algorithm returns a point p which is guaranteed to be $(1+\epsilon)$-approximate nearest neighbor of q. In other words, instead of finding the nearest neighbor of q, the algorithm returns a point p such that $D(p,q) \leq (1+\epsilon)D(p_i,q)$ for all $p_i \in C$. And the main advantage of LSH is that (after the initial preprocessing time subsequent) query processing times are guaranteed to be sublinear — $O(\sqrt{n})$ for a 2-approximate nearest neighbor.

5.3 Related work

5.3.1 Location based systems for social support

Since the emergence of online social networks, people have been seeking help from their social contacts. They can easily post their requests on Facebook or Twitter, or ask for help from strangers through online groups like “strangers helping strangers” [8]. A drawback of these schemes is that users are required to expose their requests to the public at large including those who are not in a position to offer help. In SamaritanCloud, a request message is only forwarded to people who are physically at a location where they could potentially fulfill the help request.

By 2008, geolocation technologies such as mobile phone tracking became available and were integrated with Wi-Fi, 3G and GPS navigation to enable location-based online social services – geosocial networking offers users a platform to interact relative to their locations. Yelp [9] is one of the most influential review websites, it collects people’s opinions of the places they’ve been to. Facebook Places [3] lets people share their locations to friends or public. Gowalla [5] and Foursquare [4] offer mobile applications, through which people can note their locations, recent activities and “check-in” at venues to earn awards. Some other applications [68, 120] match users with their interested locations. SamaritanCloud is different from these services in two aspects. First, SamaritanCloud serves a different purpose. Instead of looking for a social contact or crowd sourcing, users of SamaritanCloud can get on-time *physical* help for emergencies in real life. Second, SamaritanCloud offers service in a secure and private way. In existing geosocial services, users are either required to provide their real identity information (Yelp) or expose their real locations to servers (Foursquare), which leaves the open to the possibility of a privacy leak. SamaritanCloud does not require any private information from users, it uses a homomorphic encryption style scheme to match people’s locations and requests, in such a way that even the server has no clue of users’ locations.

5.3.2 Location privacy

As introduced in section 5.2, many LBS applications fetch user locations and send that information to a location-based database server. Examples of these applications include store finders and traffic reports. The increase of LBS applications makes protecting personal location information a major challenge. There are several approaches for protecting location privacy in the extant literature :

- *Middleware:* Geopriv [35] and LocServ [96] describe an approach for secure location transfer by introducing a middleware that lies between location-based applications and location tracking servers. However, the practicality of such systems have yet to be determined, as the trustworthiness of middleware is hard to prove.

- *Dummies or Anonymity:* Instead of sending out the exact location, a user sends n different locations to the server with only one of them being true [76]. The drawback of this scheme is that over the long run the server is able to figure out the user’s location by comparing the intersection of uploaded locations. Some other schemes [24, 126] separate location information from other identifying information and reports anonymous locations to the server. However, guaranteeing anonymous usage of location-based services requires that the precise location information transmitted by a user cannot be easily used to re-identify the subject.

- *Location perturbation:* another common scheme is to "blur" the exact location information to a spatial region [45, 54, 61, 94] or sending a proxy landmark instead [67]. This scheme is not suitable for the applications which require accurate locations.

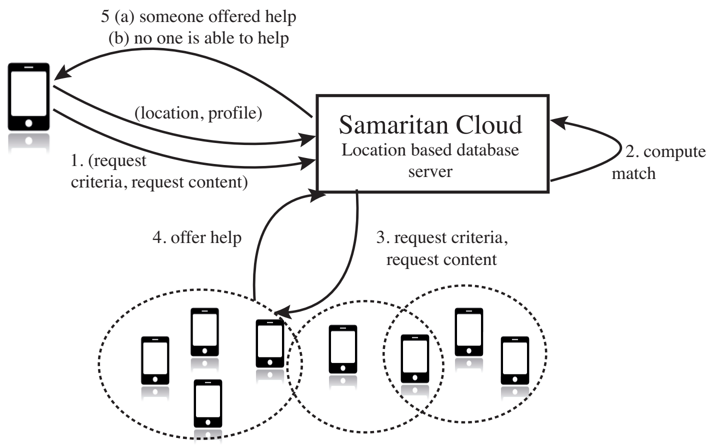
- *Location hiding:* an user can define sensitive locations where he does not want to be tracked, and the location-based application stops tracking when the user gets close to those areas [62].

As explained in the following section, the above schemes are either unsuitable or inadequate for SamaritanCloud.

5.4 System design

5.4.1 Objective

Figure 5.1: Transaction in public-mode SamaritanCloud.



The objective of SamaritanCloud is to provide a new architecture that enables people to benefit from location-based social networks not just for crowd sourcing, but also physical and cyber-physical person-to-person help. It offers users remote hands and eyes access to somewhere where they are not.

The SamaritanCloud service is deliverable to the user through a mobile application. At a high level, each user that joins SamaritanCloud is associated with an ID and a profile. Users can set different privacy levels for their profile and connect with other users as "friends". When an user needs help, he sends a message with request criteria to the closest server, with or without encryption. Then the server looks for candidates among the users that are online at the time and then multicasts the request to the candidates. A typical conversation in "public mode" (where there is no requirement to hide anything from the server) is shown in Figure 5.1.

The challenge arises in "secure mode", where the trustworthiness of the server is in question and thus the server is expected to perform the same *match* functionality without learning clients' private information. Matching profile and request (notification) with privacy preservation is also the main objective of confidential publish/subscribe systems [106], but SamaritanCloud has some differing requirements:

- In secure pub/sub schemes, typically, the change of profile is not hidden from the server; in SamaritanCloud, the change of profile attributes (e.g. location, age) should be protected.
- In pub/sub schemes, typically only (in)equality comparison is required (e.g. Keyword match); in SamaritanCloud, the server needs to compute the closeness of attributes without knowing the exact values.
- In SamaritanCloud, the profile may include high dimensional attributes and thus the server computation work load could become unreasonably high.

In this section, we first try to solve the above problems on top of existing pub/sub schemes and discuss the drawbacks. Then we propose a new scheme to address the those concerns.

5.4.2 Data type and communication protocols

We start with defining data types and protocols used between client and server.

authentication: as with most mobile applications, a client signs in to the SamaritanCloud service with a unique username and password; a TCP connection is maintained between the client and the server.

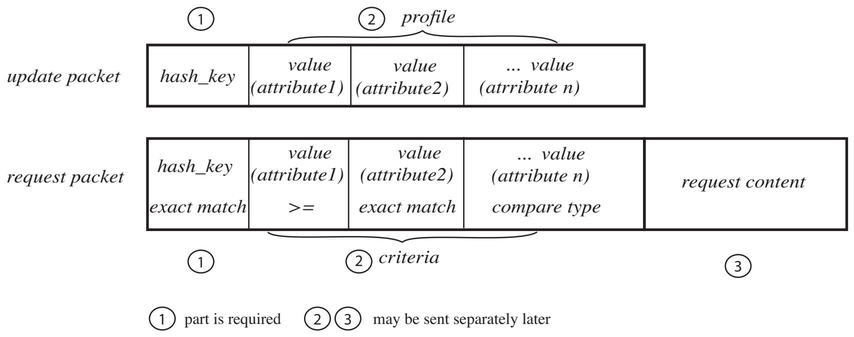
update/request: the client-side application periodically sends update messages to the server; when a client looks for help, he sends a request to the server, which disseminates the request efficiently to all the clients that have matching profiles.

notification: once the server finds a candidate for a request, it forwards the request to the client as a "notification" and waits for the response. If the client responds with an "offer help" then the server remembers this candidate for future follow-up.

follow-up: SamaritanCloud provides a simple client evaluation scheme by sending follow-up notification to the candidates who committed to offer help. We defer the more complete implementation and associated for future work. In this thesis we are primarily concerned with the problem of "secure match".

packet: the data packets used in update and request are shown in Figure 5.2. Both update and request packets include at least an LSH key. The attributes are defined as *(string, value)* pairs. In request packet, each attribute is associated with an expected comparison type (e.g. equality or inequality match).

Figure 5.2: Data packet used in SamaritanCloud.



5.4.3 Querying process

Privacy-aware querying process with pub/sub scheme

In content-based publish/subscribe model the interests of subscribers are stored in a content-based forwarding server, which guides routing of notifications to interested subscribers. We present a straight-forward way to build secure mode Samaritan-Cloud on top of existing confidential content-based publish/subscribe algorithms.

We borrow the equality and inequality comparison schemes defined in [106]. For each attribute i with inequality format, we choose l_i points, $p_1, ..., p_{l_i}$ as reference points.

Let v be the attribute value and F be an encryption function. Then the algorithm for equality comparison is :

Keygen(t) : in a departure from [106] K is generated periodically and is shared among clients.

ProfileUpdate(v) : return $F_K(p)$ where p is the closest reference point to v.

Request(v') : select r at random, return $(r, F_{F_{K(p')}}(r))$ where p' is the closest reference point to v'.

Match(v, v') : return 1 if $F_{F_{K(v)}}(r) = F_{F_{K(v')}}(r)$.

As the server can only compare (in)equality on encrypted messages, to find more candidates, a requester needs to submit a range match on certain criteria. For example, to find all candidates within distance c, a requester submits a set of encrypted values for attribute "location", covering all the reference points within $(loc \pm c)$.

Inequality comparison

Some attribute values are represented in inequality format. For example, a request with attribute (*time, t,* >=) means the request costs at most t minutes, a profile with attribute *(time, t')* means the client would love to offer help if it costs at most t' minutes, and so the client would be a match for the request if $t' >= t$.

The Inequality scheme is :

Request(v') : select r at random, return set $\{r, F_{F_K(p_i)}(r)\}$ where $p_i >= v'$.
Match(v, v') : return 1 if $F_{F_K(v)}(r) \in F_{F_K(v')}(r)$ with binary search.
The above scheme partially solves our concern – by changing the encryption key from time to time and updating profiles periodically, the server is not aware of any change in client profiles and requests. The server also can not compare the requests at different time as the encrypted messages are different. The main drawbacks of the above scheme are :

- Distribution of global key happens from time to time. This generates a lot of traffic among users and is problematic. If a client misses key generation then he can not use the service before the next key is generated. His profile and request will not be matched correctly as he uses a different key from the others.

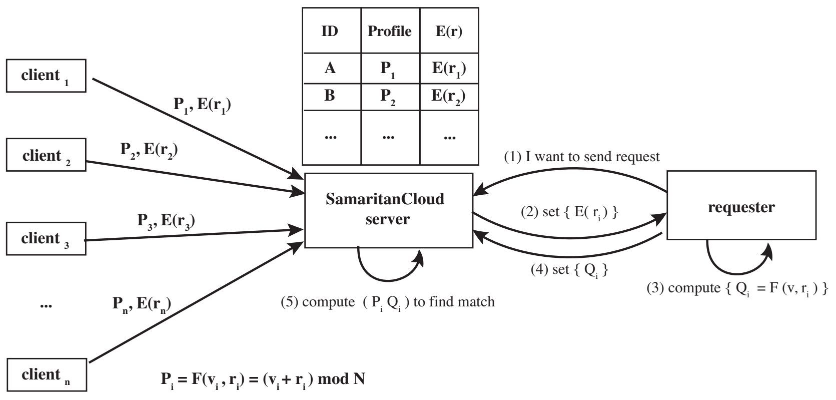
- The above scheme only computes (in)equality but not the closeness of two attributes. Take location attribute as example, to find close candidates, the requester would need to compute encrypted messages for all close locations and send them to the server as a range match. If this range is too small then no candidate may be found; while if this range is too large, the request will be broadcast to a lot of candidates.

Therefore, the existing pub/sub algorithm does not perfectly suit the requirements of SamaritanCloud. We present a new scheme - the Personalized Profile Blur Scheme or PPBS.

Personalized Profile Blur Scheme

To allow the server to compute closeness of attributes without knowing the content of client profile and request, we define the following scheme :
Let v be an attribute value,
Keygen(e, d) : keypair shared among users.
Prime N : a prime which is larger than $upper - bound(v)$.

Figure 5.3: Querying process in SamaritanCloud with secure mode.



ProfileUpdate(v) : return ((v+r) mod N, $E_e(r)$) where r is randomly chosen between $[1, P-1]$.

Request(v') : return set {(v' + r) mod N} where r is the random number generated by a candidate.

Match(v, v') : return 1 if distance((v+r) mod N, (v'+r) mod N) < threshold.

The querying process is illustrated in Figure 5.3. When a client updates his profile to the server, the profile value v_i is blurred to $(v_i + r_i)\, mod\, N$ where r_i is randomly generated each time. Encrypted r_i is also saved on the server. When a client wants to send a request, he first contacts the server to get a set of encrypted r_i generated by online users, then he computes $(v' + r_i)\ mod\ N$ and sends back to the server. The server does computation on ($(v_i + r_i)\, mod\, N$, $(v' + r_i)\, mod\, N$) and decided if $client_i$ is a candidate for the request. The correctness of this scheme is based on

$$(v\ +\ r) \equiv (v'\ +\ r)\ (mod\, P)$$

iff $v = v'$. As each client profile is blurred by a personalized random number, the server can not figure out the closeness between two clients' profiles. As the random number is generated each time when the client updates profile, the profile change is hidden from the server too. The advantages of this scheme over the pub/sub encryption scheme of [106] are :

- profile and request are blurred with a random number generated by each client; generation of encryption key is only needed at the beginning. This offers more flexibility to clients (without worrying about missing key generation) and reduces traffic from key generation.

- profile and request are not encrypted with encryption keys, which allows the server to compute closeness between a profile and a request; however, the server can not learn the closeness between two profiles as they are blurred with different random numbers.

This scheme fully addresses our security concerns of querying process in secure mode. We discuss performance optimization in section 5.6.

5.4.4 Request forwarding

In public mode, request content is attached at the end of a request packet and would be forwarded to the candidates by the server. In secure mode, once the server finds out candidates who would offer help, the request content would be encrypted with those clients' public key and then forwarded to them. However, when a client sends out request, since he does not know who could offer help and whose public key should be used for encryption, the request content is not included in the request packet. We reply on a trustable Lightweight Directory Access Protocol (LDAP) server to maintain the directory of public keys, which are in the form of certificates. Once the server finds out client B is a candidate to help client A, client A gets B's public key from LDAP server and encrypts his request, then the server forwards the encrypted request to client B, B decrypts it with his private key.

5.5 Evaluation

5.5.1 Implementation

The implementation of SamaritanCloud includes two parts – client side application and the server. We implemented the client side application with a prototype mobile application running on iOS 5.0, allowing users to

- log in or register with a unique user name and password.
- choose timer interval to update profile; by setting a fixed update time interval, exact profile change time is not exposed to the server.
- input request location (either latitude and longitude coordinates or zipcode) and request content.
- get notification when the client is close to a requested location; respond with willingness to offer help.
- get notification if anyone offers help to the client's request.
- get follow-up questionnaire of (1) "did you offer the help that you committed to?" if the client was a candidate, (2) "was the request resolved?" if the client was the requester.

We implemented the server using the Python library. The major concerns of SamaritanCloud performance are client application battery consumption and server scalability. We examine those two concerns with experimental results.

5.5.2 Mobile application

We first examine SamaritanCloud on mobile devices. Our application registers itself as a background VoIP service allowing it to (1) maintain a persistent connection with the server, (2) periodically fetch location information, even if the app is running in the background. Our application utilizes *Core Location Framework* to get the physical location. Location coordinates can be obtained through (a) standard location service and (b) significant-change location service. With standard location service, location data is gathered with either cell triangulation, Wi-Fi hotspots or GPS satellites, depending on required accuracy. As our application periodically updates location to the server, it saves battery life by enabling the standard location service shortly as per the interval defined by client[1]. With significant-change location service, a location updates is triggered only if a device moves from one cell tower to another, thus it provides only coarse location update. There is an accuracy-power tradeoff — having GPS location update is accurate but power consuming while significant-change location service provides a low-power approach. To test the worst possible impact of our application on battery life, we run the applications pinned to the foreground with an intensive frequency of updates (once per minute) via both 3G and WiFi. In each update, the mobile application fetches location coordinates (lat, lon), computes $((lat + r)\, mod\, P, (lon + r)\, mod\, P)$ (P is a prime with 64 bits, r is randomly chose between [1, P]), sends the result together with E(r) (E is 512 bit RSA function) to the server. The results in Table 5.1 demonstrate that the impact on battery life is acceptable, as in real world deployment the application would run in background most of the time, and location updates would be less frequent.

Table 5.1: Impact of SamaritanCloud iOS application on battery life.

network	standard location service (GPS)	standard location service (WiFi/cellular)	significant-change location service
3G	10h	12h 8mins	12h 10mins
WiFi	11h 14mins	16h 14mins	16h

5.5.3 Server scalability

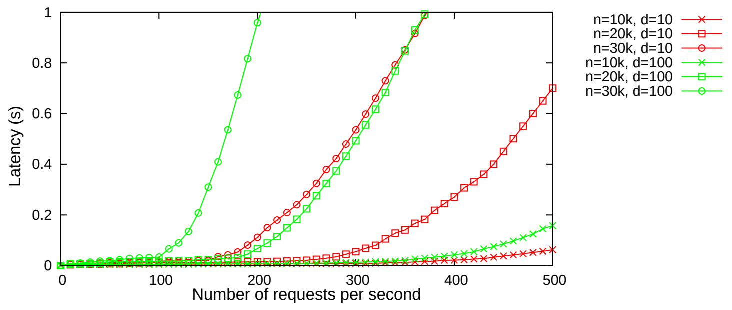
For server side performance we are primarily interested in understanding the rate of *requests* that a single server can handle, as this serves as the dominating factor controlling the number of online users that the server can support. In public mode, messages are sent in plain-text format. As group information is not a concern, client profiles are saved in hash tables. Therefore, fetching profiles and computing $match$ costs $O(dk)$ operations where d is the number of attributes and k is the

[1]Since fetching location data requires a few seconds, the first location coordinates returned are usually saved from the last time. Timestamp of the location data is used to determine if the update is fresh.

average number of profiles in the same hash bin. In this case, server performance should be consistent when number of clients increases. However, in secure mode, matching d attributes with n clients costs (dn) operations. We focus on server performance when the value of d and n increase.

Searching time is measured starting from when the requester sends out blurred attributes till he gets a response from the server, using the function $datetime.datetime.now()$ available in Python, with the accuracy of microseconds. We ran our server implementation on a machine with 2.5 GHz Intel Core i5 and 4GB memory. To minimize the affect of Internet latencies and instabilities, we simulated the server and clients on two machines in the same LAN.

Figure 5.4: Profile matching latency



Server scalability in secure-mode is shown in Figure 5.4. When profiles include low dimensional attributes ($d \leq 10$), the server is able to match 200 requests against 30,000 profiles per second before more than 100ms of latency is incurred. When profiles include high dimensional attributes ($d \approx 100$), the server is able to match 120 requests against 30,000 profiles per second before more than 60ms of latency is incurred. To protect the system from DoS attacks, the server only handles one request from the same client within 5 minutes, the average request rate with 30k users is 100 requests per second. Therefore, the server can serve around 30,000 online users. It is also worth noting that a highly-optimized implementation of the server is likely to support the same amount of users with much greater performance.

5.6 Discussion

Let d be the number of attributes in profile. For each update, a client sends d attributes to the server. For each request, a requester calculates “blurred” values for d attributes with n different random numbers, which is $O(dn)$. Once the server receives the request, it computes $match$ with $O(dn)$ operations. The larger d is, the more the traffic and work load in the system.

There are two approaches to reduce searching complexity :

- The server maps clients into hash tables and compares a client’s profile with a request only if the client is in the same bucket as the requester.
- Reducing d dimensional data into low dimensional data without losing locality.

The first approach improves searching efficiency only if the server knows the groups of client profiles in advance. However, this exposes the similarity between clients to the server. The second approach was studied by LSH algorithms. LSH schemes preprocess all the n points as described below :

- Define a new family G of hash functions g, where each g is obtained by concatenating k randomly chosen hash functions h_i.
- Construct L hash tables, each corresponding to a different randomly chosen hash function g.
- Hash all n points into each of the L hash tables.

As we do not want to expose profile content to the server, the preprocessing step has to be done on the client side. Each client calculates L hash values then sends the blurred results to the server. The following process is the same as the scheme discussed in section 5.4.3. The only difference is that instead of sending dn blurred attributes, a client updates L blurred hash values to the server. [69] proposed an LSH routine to offer 2-approximate nearest neighbor search with $O\sqrt{n}$ query time, where $L = O(\sqrt{n})$. Then the server spends $O(n\sqrt{n})$ time on storing client profiles and $O(d\sqrt{n})$ time on searching for candidates. When d gets large (e.g. n^2), the system benefits from LSH throughimproved server efficiency and a reduction in the size of data transferred between clients and the server.

5.7 Conclusion

To the best of our knowledge, SamaritanCloud is the first of its kind in using location-based social networking system to enable people to *physically* help each other. In our system, each user gets an unique username and password. When an user needs help at a particular location, the request is sent to the server and the server forwards the request to the users located near that location and who fulfill several other criteria associated with the request. The users near the area of interest are looked up efficiently using LSH. We have also looked into internal and external security attacks. To hinder external security attacks SamaritanCloud uses standard public key technology. So, the only possible internal security risk is when SamaritanCloud server is compromised. We avert this risk by sending the blurred location and not the actual location of an user to the server. We have implemented a SamaritanCloud mobile application for iOS 5.0. The application uses two types of location updates, a)standard location service and b) significant-change location service. Standard location service updates the SamaritanCloud server about the location of the device with the help of one of of cell triangulation, Wi-Fi hotspot or GPS. Significant-change location service, updates the server about the update of location if the device moves from one cell tower to another thus giving a course location update. Because of the limited battery-life and processing power of smartphones, the mobile application allows users to manually select the frequency of location update and the level of security that they desire. Our SamaritanCloud system, opens up an entirely new approach to enable people to benefit from location-based social networks.

Bibliography

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[2] AMD Virtualization Technology. `http://www.amd.com/`.

[3] Facebook places. `http://www.facebook.com/facebookplaces`.

[4] Foursquare. `https://foursquare.com/`.

[5] Gowalla. `http://gowalla.com/`.

[6] Microsoft Azure Services Platform. `http://www.microsoft.com/azure/default.mspx`.

[7] Microsoft Windows Azure Platform. `http://www.microsoft.com/azure/default.mspx`.

[8] Strangers helping strangers. `http://www.facebook.com/SHStrangers`.

[9] Yelp. `http://www.yelp.com/`.

[10] Method and system for protecting websites from public internet threats. United States Patent 7,260,639, August 2007.

[11] Amazon Web Services: Overview of Security Processes, 2008. `http://developer.amazonwebservices.com`.

[12] Method and system for providing on-demand content delivery for an origin server. United States Patent 7,376,736, May 2008.

[13] Securing the federal government's domain name system infrastructure, 2008. `http://www.whitehouse.gov/omb/memoranda/fy2008/m08-23.pdf`.

[14] 4,000,000,000 $\ll$ flickr blog. `http://blog.flickr.net/en/2009/10/12/4000000000/`.

[15] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS 2006*, 2006.

[16] P. K. Agarwal and J. Matousek. Ray shooting and parametic search. *SIAM Journal on Computing*, 1993.

[17] Alexa top 500 global sites. `http://www.alexa.com/topsites`.

[18] N. Alon and J. Spencer. *The Probabilistic Method*. John Wiley and Sons, Inc, 2008.

[19] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 2000.

[20] M. Armbrust, A. Fox, and etc R. Griffith. A view of cloud computing. *Communications of the ACM*, 2010.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.

[22] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. 2004.

[23] I. Beijnum. *BGP*. O'Reilly, 2002.

[24] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2003.

[25] C. Boulton. Novell, Microsoft Outline Virtual Collaboration. *Serverwatch*, 2007.

[26] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE International Conference on Computer Communications (INFOCOM)*, New York, New York, March 1999.

[27] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. Peerson: P2p social networking – early experiences and insights. In *International Workshop on Social Network Systems (SNS)*, Nuremberg, Germany, March 2009.

[28] M. Cha, H. Kwak, P. Rodriguez, Y. Ahn, and S. Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *IMC*, 2007.

[29] M. Cha, A. Mislove, B. Adams, and K. P. Gummadi. Characterizing social cascades in flickr. In *WOSN*, 2008.

[30] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *IWQoS*, 2008.

[31] L. Cherkasova, D.Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETERICS Performance Evaluation Review*, 2007.

[32] L. Cherkasova, D.Gupta, and A. Vahdat. When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments. 2007.

[33] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall PTR, 2007.

[34] A. Clauset, C. R. Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM Review*, 2009.

[35] A. Cooper and T. Hardie. Geopriv: creating building blocks for managing location privacy on the internet. *IETF Journal*, 2002.

[36] G. Cormode and B. Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), June 2008.

[37] Symantec Corporation. Symantec gateway security products dns cache poisoning vulnerability. `http://securityresponse.symantec.com/avcenter/security/Content/2004.06.21.html`.

[38] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, New York, October 2003.

[39] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using a peer-to-per lookup service. 2002.

[40] D.Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *ACM/IFIP/USENIX Middleware*, 2006.

[41] Diaspora*. `http://www.joindiaspora.com/`.

[42] R. Dittner and D. Rul. *The Best Damn Server Virtualization Book Period*. Syngress, 2007.

[43] Dnssec.net. Dns security extensions securing the domain name system. `http://www.dnssec.net`.

[44] C. Douligeris and A. Mitrokotsa. Ddos attacks and defense mechanisms: classification and state-of-the-art. In *Computer Networks*, 2004.

[45] M. Duckham and L. Kulik. A formal model of obfuscation and negotiation for location privacy. *Pervasive*, 2005.

[46] Facebook. Facebook. `http://www.facebook.com`.

[47] Facebook and youtube dominate workplace traffic and bandwidth. `http://www.scmagazineuk.com/facebook-and-youtube-dominate-workplace-traffic-and-bandwidth/article/168082/`.

[48] Facebook statistics. `http://www.facebook.com/press/info.php?statistics`.

[49] M. J. Freedman. Experiences with coralcdn: A five-year operational view. In *Symposium on Networked System Design and Implementation (NSDI)*, San Jose, California, April 2010.

[50] M. J. Freedman, E. F., and D. Mazieres. Democratizing content publication with coral. In *Symposium on Networked System Design and Implementation (NSDI)*, San Francisco, California, March 2004.

[51] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *CCS*, 2002.

[52] S. Friedl. An illustrated guide to the kaminsky dns vulnerability, 2008. `http://unixwiz.net/techtips/iguide-kaminsky-dnsvuln.html`.

[53] S. Garriss, M. Jaminsky, M. J. Freedman, B. Karp, D. Mazires, and H. Yu. Re: Reliable email. In *Symposium on Networked System Design and Implementation (NSDI)*, San Jose, California, May 2006.

[54] B. Gedik and L. Liu. A customizeable k-anonymity model for protecting location privacy. In *Proceeding of the International Conference on Distributed Computing Systems*, 2005.

[55] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge. In *ACM/USENIX Internet Measurement Conference (IMC)*, San Diego, California, October 2007.

[56] Google+. Google+. `https://plus.google.com`.

[57] S. Govindan, A.R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *ACM VEE*, 2007.

[58] I. Green. Dns spoofing by the man in the middle. `http://www.sans.org/reading_room/whitepapers/dns/dns-spoofing-man-middle_1567`.

[59] A. G. Greenberg, P. Flajolet, and R. E. Ladner. Estimating the multiplicities of conflicts to speed their resolution in multiple access channels. *J. ACM*, 34(2):289–325, 1987.

[60] A. G. Greenberg and S. Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *J. ACM*, 32(3):589–596, 1985.

[61] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloacking. In *Proceedings of the International Conference on MobiSys*, 2003.

[62] M. Gruteser and X. Liu. Protecting privacy in continuous location-tracking applications. *IEEE security and privacy*, 2004.

[63] T. Guo and J. Chen. Spoof detection for preventing dos attacks against dns servers. 2006.

[64] C. Gutierrez, R. Krishnan, R. Sundaram, and F. Zhou. Hard-dns: Highly-available redundanty distributed dns. *Military communications conference*, 2010.

[65] K. Haugsness and the ISC Incident Handlers. Dns cache poisoning detailed analysis report verison 2. `http://isc.scans.org/presentations/dnspoisoning.php`.

[66] Hitwise intelligence - robin goad - uk. `http://weblogs.hitwise.com/robin-goad/2010/08/facebook_accounts_for_1_in_6_uk_page_views_has_it_reached_saturation_point.html`.

[67] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *Proceedings of the International Conference on Mobile Systems*, 2004.

[68] Q. Huang and Y. Liu. On geo-social network services. *Geoinformatics*, 2009.

[69] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimenionality. In *ACM STOC*, 1998.

[70] B. Jansen, H. V. Ramasamy, and M. Schunter. Policy enforcement and compliance proofs for Xen virtual machines. In *ACM VEE*, 2008.

[71] J. Jiang, C. Wilson, X. Wang, P. Huang, W. Sha, Y. Dai, and B. Y. Zhao. Understanding latent interactions in online social networks. In *IMC*, 2010.

[72] A. N. Joinson. Looking at, looking up or keeping up with people?: Motives and use of facebook. In *CHI*, 2008.

[73] Y. W. Ju, K. H Song, E. J. Lee, and Y. T Shin. Cache poisoning detection method for improving security of recursive dns. *The 9th International Conference on Advanced Communication Technology*, 2007.

[74] Kaminsky. D. doxpara research. `http://www.doxpara.com`.

[75] N. Kennedy. Facebook's photo storage rewrite. `http://www.niallkennedy.com/blog/2009/04/facebook-haystack.html`.

[76] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *Proceddings of IEEE International Conference on Pervasive Service*, 2005.

[77] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *ACM VEE)*, 2009.

[78] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM STOC*, 1997.

[79] J. Komlós and A. G. Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.

[80] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM J. COMPUT*, 2000.

[81] F. Leighton. Internet security insight. `http://www.akamai.com/html/perspectives/insight_tl_internet_security.html`.

[82] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cyle. In *KDD*, 2009.

[83] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, and A. Tomkins. Geographic routing in social networks. *PNAS*, 2005.

[84] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX Annual Technical Conference (USENIX)*, San Antonio, Texas, June 2003.

[85] C. Liu. and P. Albitz. *DNS and BIND*. O'Reilly, 2006.

[86] S. Marti, P. Ganesan, and H. Garcia-Molina. Dht routing using social links. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, La Jolla, California, February 2004.

[87] J. MatouÅąek. Reporting points in halfspace. *Computational Geromery: Theory and Applications*, 1992.

[88] S. McCanne and C. Torek. A randomized sampling clock for cpu utilization estimation and code profiling. In *USENIX*, 1993.

[89] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 1993.

[90] Carnegie Mellon. Cert advisory ca-1997-22 bind - the berkeley internet name daemon. `http://www.cert.org/advisories/CA-1997-22.html`.

[91] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *WOSN*, 2008.

[92] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, 2007.

[93] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: Inferring user profiles in online social networks. In *WSDM*, 2010.

[94] M. F. Mokbel, C-Y Chow, and W. G. Aref. The new casper: a privacy-aware location-based database server. *IEEE 23rd International Conference on Data Engineering*, 2007.

[95] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *ACM VEE*, 2008.

[96] G. Myles, A. Friday, and N. Davies. Preserving privacy in environments with location-based applications. *Pervasive Computing, IEEE*, 2003.

[97] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS 2009*, 2009.

[98] S. Basagiannis N. Alexiou, P. Katsaros, T. Dashpande, and S. Smolka. Formal analysis of the kaminsky dns cache-poisoning attack using probabilistic model checking. In *IEEE 12th International Symposium on High-Assurance System*, 2010.

[99] CNET News. Phishers using dns servers to lure victims? `http://news.cnet.com/Phishers-using-DNS-servers-to-lure-victims/2100-7349_3-5604555.html`.

[100] M. Olney, P. Mullen, and K. Miklavcic. Dan kaminsky's 2008 dns vulnerability. In *Sourcefire Vulnerability Report*, 2008.

[101] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in a virtual machine monitor. In *ACM VEE*, 2008.

[102] K. Park, Z. Wang, V. Pai, and L. Peterson. Codns:maskingdns delays via cooperative lookups. 2004.

[103] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee. Wsec dns: Protecting recursive dns resolvers from poisoning attacks. *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009.

[104] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving content delivery using provider-aided distance information. In *IMC*, 2010.

[105] B. C. Popescu, B. Crispo, and A. S. Tanenbaum. Safe and private data sharing with turtle: Friends team-up and beat the system. In *International Workshop on Security Protocols (IWSP)*, Cambridge, United Kingdom, April 2004.

[106] C. Raiciu and D. S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. *Securecomm and Workshops*, 2006.

[107] V. Ramansubrmanian and E. G. Sire. Perils of transitive trust in the domain name system. In *Internet Measurement Conference*, 2005.

[108] V. Ramasubramanian and E. Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM*, 2004.

[109] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*, 2009.

[110] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *International Middleware Conference (Middleware)*, Heidelberg, Germany, November 2001.

[111] S. Rueda, Y. Sreenivasan, and T. Jaeger. Flexible security configuration for virtual machines. In *ACM workshop on Computer security architecures*, 2008.

[112] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. In *IEEE S&P*, 1998.

[113] H. Samet. The design and analysis of spatial data structures. Addison-Wesley, 1990.

[114] S. Sarat, V. Pappas, and A. Terzis. One the use of anycast in dns. In *International Conference on Computer Communications and Networks*, 2006.

[115] S. Scellato, C. Mascolo, M. Musolesi, and J. Crowcroft. Track globally, deliver locally: Improving content delivery networks by tracking geographic social cascades. In *WWW*, 2011.

[116] C. Schuba and E. Spafford. Addressing weakness in the domain name system protocol. Purdue University, Dept. of Computer Science, 1994.

[117] S. Siddha, V. Pallipadi, and A. V. D. Ven. Getting maximum mileage out of tickless. In *Linux Symposium*, 2007.

[118] M. Snyder, R. Sundaram, and M. Thakur. A game-theoretic framework for bandwidth attacks and statistical defenses. In *IEEE LCN*, 2007.

[119] J. Sobel. Scaling out. `http://www.facebook.com/note.php?note_id=23844338919&id=9445547199`.

[120] D. Stackpole. Dynamic geosocial neworking, 06 2008.

[121] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flash crowds. In *ICNP*, 2002.

[122] J. Stewart. Dns cache poisoning - the next generation. *Computer and Information Science*, 1997.

[123] R. Sundaram, M. Snyder, and M. Thakur. Preprocessing dns log data for effective data mining. In *Proceedings of ICC*, 2009.

[124] Microsoft Support. Description of the dns server secure cache against pollution setting. `http://support.microsoft.com/kb/316786`.

[125] D. Talbot. Vulnerability seen in amazon's cloud computing. *MIT Tech Review*, October 2009.

[126] K. P. Tang, P. Keyani, J. Fogarty, and J. I. Hong. Putting people in their place: An anonymous and privacy-sensitive approach to collecting sensed data in location-based applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 2006.

[127] J. Terrace, H. Laidlaw, H. Liu, S. Stern, and M. J. Freedman. Bringing p2p to the web: Security and privacy in the firecoral network. In *IPTPS*, 2009.

[128] M. Theimer and M. Jones. Overlook: Scalable name service on an overlay network. 2002.

[129] D. N. Tran, F. Chiang, and J. Li. Friendstore: Cooperative online backup using trusted nodes. In *International Workshop on Social Network Systems (SNS)*, Glasgow, United Kingdom, April 2008.

[130] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *16th USENIX Security Symposium*, 2007.

[131] Twitter. Twitter. `http://www.twitter.com`.

[132] P. Vajgel. Needle in a haystack: Efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`.

[133] R. P. Weicker. Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *SIGPLAN Notices*, 1988.

[134] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *ACM VEE*, 2009.

[135] C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, 2009.

[136] M. P. Wittie, V. Pejovic, L. Deek, K. C. Almeroth, and B. Y. Zhao. Exploiting locality of interest in online social networks. In *CoNEXT*, Philadelphia, Pennsylvania, December 2010.

[137] Haiyong Xie, Y. Richard Yang, Arvind Krishnamurthy, Yanbin Liu, and Avi Silberschatz. P4p: Provider portal for applications. In *SIGCOMM*, 2008.

[138] Yahoo! Yahoo! geocities. `http://en.wikipedia.org/wiki/GeoCities`.

[139] A. C. Yao and F. F. Yao. A general approach to d-dimensional geometric queries. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, 1985.

[140] L. Yuan, K. Kant, P. Mohapatra, and C. Chuah. Dox: A peer-to-peer antidote for dns cache poisoning attacks. *IEEE International Conference on Communications*, 2006.

[141] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud. *NCA IEEE Computer Society*, 2011.

[142] F. Zhou, L. Zhang, E. Franco, A. Mislove, R. Revis, and R. Sundaram. Webcloud: Recruiting social network users to assist in content distribution. *under submission to NCA IEEE Computer Society*, 2012.

[143] M. Zink, K. Suh, Y. Gu, and J. Kurose. Watch global, cache local: Youtube network traffic at a campus network - measurements and implications. In *Conference on Multimedia Computing and Networking (MMCN)*, San Jose, California, January 2008.