Contributions to Content Placement, Load-balancing and Caching: System Design and Security

Christoph Neumann

Contributions to Content Placement, Load-balancing and Caching: System Design and Security

Christoph Neumann

Habilitation à Diriger des Recherches
Université de Rennes 1 - 2016

Rapport d'activité

Jury:

Imad Aad, Data Analyst, Swisscom, Suisse	Examinateur
Olivier Festor, Professeur, Université de Lorraine, France	Rapporteur
Isabelle Guérin Lassous, Professeur, Université Lyon 1, France	Rapporteur
Laurent Mathy, Professeur, Université de Liège, Belgique	Rapporteur
Guillaume Pierre, Professeur, Université de Rennes 1, France	Président
Gwendal Simon, Maître de Conférences, TELECOM Bretagne	Examinateur

Abstract

Distributed systems invaded our daily lives offering a large set of services and applications such as social networks, chats, video consumption etc. Despite the large diversity of services offered, the underlying systems have many things in common. They must handle a large set of computing and storage nodes and must decide where to place the data and how to balance the load between the nodes. Therefore, these systems implement some *placement* and *load-balancing policies*. Data caches are used to further optimize access to data. The *caching policy* - a specific instance of the placement policy - decides which items to keep in cache and which ones to evict. Even small scale and dedicated architectures such as hard disk drives or CPUs employ the above policies and mechanisms.

This document presents several contributions to content placement, load-balancing and caching policies. We consider two different aspects: (i) the system design, *i.e.*, the design and evaluation of such policies for different architectures and (ii) the security, *i.e.*, understanding how to exploit the caches, either for an attack or to build security applications.

We start by presenting several contributions around the system design of such policies for Video-on-Demand services. We first consider a system where the video content is pro-actively placed on home gateways managed by an operator. During content consumption the gateways fetch the content from other home gateways. In contrast, the second system we study exclusively exploits the cloud. It serves targeted videos and must decide which content to cache instead of recomputing it at each user request. The pay-per use model of the cloud requires new and specific attention when designing the caching policies. We then consider security applications and attacks of caches. We build a trusted time stamping service using the caches of the publicly available Domain Name Service. We also design and demonstrate attacks on the caches of modern CPUs, allowing us to pass information from one process (or virtual machine) to another despite the isolation. Finally, we conclude with an outlook and some future work in the domain of caching and load-balancing.

Contents

1

Foreword

This document summarizes some of my research since I defended my PhD in 2005. Working in the industry, I had the occasion to work on a very diverse set of applications and infrastructures. From an applicative point of view, I worked on Video on Demand content distribution systems, targeted advertisement, data analytics for network operators and security. From an infrastructure point of view, I designed systems that build upon home gateways as well as systems that rely on cloud infrastructure. In terms of research domains, my contributions since after my PhD relate to networking, distributed systems and security. Despite this large diversity, a set of research issues are common to all of the above systems and this documents resumes this common denominator in my research, namely *caching* and *load-balancing* mechanisms. Further, in my research I am always interested in how things work, ie. building systems and algorithms, but also how things do *not* work, ie. understanding the security of a system or algorithm and how to break or exploit it for a different usage. Therefore this document covers system design and security aspects with respect to caches and load-balancing.

Figure 1.1 presents the timeline of my research positions and research activities since after finishing my PhD. In 2006, as a postdoctoral researcher at the Thomson Paris Research Lab I worked on peer-to-peer Video-On-Demand systems relying on home gateways. Content placement and load-balancing were key aspects in the system design, and part of this work is presented in this document. During this time I was offered the opportunity to join a business unit within Thomson that would implement and push for commercial deployment of this system. Since I am always eager to build things that have

Figure 1.1: Timeline of research positions, research topics.

impact and would eventually end-up in a product or in an actual deployment, I accepted this opportunity. Therefore, during two years - from 2007 to 2009 - I quit research and worked as a product line and project manager The entity I worked in also developed video content tracking solutions relying on video and audio watermark technologies. In this context, I also led the development of content marking and tracking systems which download, identify, track and filter video and audio contents on the Internet. This gave me the opportunity to work on security aspects - eventhough not from a research perspective.

In 2009, I came back to research and joined the Technicolor Security Labs (Thomson has rebranded itself into Technicolor). The initial focus of my work

at this time was around home network security monitoring. The focus was on extracting security relevant events for the end-user. I also designed and developed device fingerprinting techniques that try to uniquely identify a device by observing the traffic patterns generated. Starting in 2011 the focus of my work shifted from the home network to the cloud - the cloud is also becoming one of the trending research topics at that time. The notions of large scale data collection and analytics using the cloud also just started at this time, and that is why I particularly looked at the security of cloud platforms and co-supervised the PhD of Clémentine Maurice on that subject. Interestingly, caches (of CPUs) are one important vector of attacks in the cloud, and part of this work is presented in this document.

Starting in 2012, I once again focused on video content distribution. In contrast to my previous work on video content distribution, the system exclusively relied on cloud infrastructure. Also, the system had the specificity to serve *personalized* content to the end-user. This is a particular challenge in cloud-based content distribution systems serving potentially millions of users. Caches help optimizing the cost of such a system, and I will present my contributions in this field in this document.

More recently, my research is directed at extracting valuable information for the users and operators from low-level system and network data, typically system logs or network traces collected at end-user devices such as gateways. The collected information can be sensitive and massive (millions of devices), which links back to my research about distributed systems and security, i.e. how to build scalable and secure data collection and processing systems.

The work presented in this document covers most of the above mentioned periods and research positions. Despite its large diversity a common set of challenges and research questions arose. The set of references that I use as a basis for this document are highlighted in bold in Figure 1.1. These references are also marked in bold in the bibliography.

Finally, I would like to thank all my colleagues, co-authors and especially Clémentine Maurice during her Ph.D. thesis. Without them all these contributions would not have been possible.

2

Introduction

Distributed systems invaded our daily lives offering the end-users a large set of applications such as social networks, chats, video consumption, cloud storage etc. From a user perspective the services provided by these application are very different. Still, all these systems handle a large set of computing and storage nodes and must decide where to place (and possibly duplicate) the data and how to balance the load between the nodes. Therefore, these systems implement some *placement policy*, a strategy that determines where and how to store the data. Further, the *load-balancing policy* determines where to fetch the data from or where to compute some task.

Caches are dedicated data stores that (temporarily) store some duplicate of the data. Clients can access data in the cache instead of fetching it from the origin. Caches enable to optimize some parameter of the system, e.g. access latency to data or network link usage. Tightly related to *placement polices* are the *caching policies*. Caching policies can be considered as a variant of content placement policies. In general caching policies are local to a given cache and take into account the system load that varies over time.

While the above mechanisms apply to large scale distributed systems many of these concepts also appear in smaller scale and dedicated architectures. Hard disk drives employ caches to speed-up access to files. CPUs employ caches speed-up access to memory pages. Recent CPU micro-architectures from Intel use up to 37 different caches.[1] Load-balancing and caching policies therefore also play an important roles in these architectures.

[1] On the Intel Haswell micro-architecture with 18 cores

This document presents several contribution to content placement, load-balancing and caching policies. The first two following chapters deal with the system design of such policies for Video-on-Demand services (Chapter 3 relies on home gateways and Chapter 4 relies on the cloud). Chapter 5 then considers security applications and attacks of caches. In the following, I provide some background, discuss the developments in the research community and position our contributions.

2.1 Load-balancing and data placement for video streaming

Load balancing and data placement is an inherent and old topic in distributed systems. It has been investigated many years ago in the context of job scheduling and also relates to the more general bins and balls problems [ELZ86, MTS90, MRS01, ACMR95]. The proposed algorithms were generic, *i.e.*, not cast for a specific infrastructure or application. When it became clear that video will represent an important part of Internet traffic (today 70% of the Internet traffic is video [bib]), the research community focused on the challenges related to video distribution. The first approaches for video delivery were server-based. Different data placement and load balancing policies for distributed server-based video streaming services have been proposed, e.g., relying on mirroring [BFD97] or random duplication of data blocks across storage locations [Kor97]. Solution based on network caches and video prefix caching [SRT99] have also been proposed.

When peer-to-peer (P2P) systems became popular, specific load balancing policies have been designed for these systems. A load balancing policy in a peer-to-peer consists for a peer in deciding from which peer to download data and which piece of data to download. We will refer to these P2P specific instances of load balancing policies as *pull schemes* - the peers have to decide which piece of data to *pull* and where to *pull* data from. Once pulled content has been stored locally, the peer may then in turn distribute this content to other peers. BitTorrent, the most notorious P2P file downloading protocol, implements the pull schemes "rarest first" and "optimistic unchoke", which have been largely studied in the literature [LUKM06]. A set of specific pull schemes for peer-to-peer systems providing live [CRSZ01, ZLLY05, SBI05, TK07] and Video on Demand [DLHC05, AGR06, VIF06] streaming have also been proposed. In general, the proposed pull schemes decide which pieces of data to download according to the position in the video playback - "rarest first' not being the only criteria anymore. In P2P video streaming (and in

contrast to simple file downloading), an additional difficulty is to overcome a possibly limited upstream bandwidth of serving peers while still sustaining the video playback rate. Dana et al. [DLHC05] and Tewari et. al [TK07] proposed BitTorrent-based (VoD and live respectively) streaming services where a peer can decide to pull data from additional and dedicated servers.

In the all above P2P systems, peer interest plays the central role in content transmission and storage. The peers are *selfish*, *i.e.*, a given peer only *pulls* content only if the content is of interest for itself. Once pulled content has been stored locally, the peer may then in turn distribute this content to yet other selfish peers. Such a pull-based system design is natural when the individual peers are autonomous and selfish. However, when the individual peers are under common control, for example in the case of a residential home gateway or set-top box under the control of a network or content provider, a richer range of system designs becomes possible. My contributions in chapter 3 discusses the possible design choices in such an environment for a Video on Demand application.

2.2 Caching policies

A variety of systems rely on caches: CPU caches speed-up access to memory pages; hard-disk caches speed-up access to files; network caches (e.g., HTTP proxies, Content Delivey Networks) optimize network traffic, load, cost and speed-up data access; video prefix caches (as mentioned in the previous section) decrease the startup delay of a video. Caching is also is also an intrinsic feature of routers in the Content-Centric Networking approach [JST+09]. The associated caching policies - which determine how to efficiently use cache resources - have received a lot of research attention.

Most caching policies assume fixed cache sizes and are flavors of Least Recently Used (LRU) or Least Frequently Used (LFU) policies. An item is evicted from the cache as a function of the item access frequency or the last time of access. These algorithms try to maximize the hit ratio of the cache. Most related work adapts the LRU and LFU caching policies to a specific context or application. [CI97, RV00] study HTTP proxy caching policies, considering parameters such as the item size or the cost to retrieve the item from its origin. CPU caches in general implement pseudo-LRU, a variant requiring little item tracking overhead and mimicking the behavior of LRU. Some CPUs adapt the policy according to the workload [QJP+07, JGSW]. We will discuss in chapters 5 how to exploit these CPU cache policies to build covert-channels.

ARC [MM03b] is a flavor of LRU with uniform page sizes, applicable to hard-disk drives or CPU caches. [OOW93] is as LRU policy specifically designed for disk caches in the context of database workloads. *memcached* used in many cluster or cloud deployments is limited by the RAM size and employs an LRU policy to select the data items maintained in RAM.

Time based cache policies apply a time-to-live (TTL) to every item stored in the cache [LC97]. The cache server decrements the TTL every second and evicts the item from the cache once the time-to-live reaches zero. Time based policies aim at maintaining data consistency. They do not impose a cache size limit by themselves but are often used in addition to some LRU or LFU policy that works with fixed cache sizes. Time based policies are used by DNS cache resolvers [JSBM02] and web proxies [LC97]. In general, the TTL value of an item is fixed by the origin server of the item (the authoritative DNS domain manager or the origin Web server). In Chapter 5 we show how to exploit the DNS cache resolver infrastructure to build a trusted timestamping service.

In the context of cloud applications, caching consists in temporarily storing data on some cloud storage services (e.g., Amazon S3 or Amazon ElastiCache) instead of recomputing or fetching it from another location; cloud-based caching uses cloud storage as a data cache. In theory the cloud does not impose a limit on a cache size; the size of the cache is virtually infinite as long as the user is willing to pay for it. Still, most related work also considers fixed-sized caches and implement some LRU flavor. Banditwattanawong et al. [Ban12] only consider fixed size architectures and takes into account cloud data-out traffic cost to decide which item to keep stored in a dedicated data cache. Only Yuan et al. [YYLC10] study the trade-off of cloud compute versus cloud storage in scientific workflows without imposing a limit on the cloud storage. Their proposed model is specific to the considered application and their approach does not fit a more general caching system.

The cloud infrastructure, by offering unlimited resources and a pay per use cost model, is an invitation to revisit cache policies. There is a need to define new cost effective policies that are not necessarily bounded by the cache size but rather consider the various cost factors. I present several caching policies for the cloud in Chapter 4.

2.3 Cache-based attacks

In modern x86 micro-architectures, the CPU cache is an element that is shared by cores of the same processor (a processor is composed of several cores). Isolation prevents processes from directly reading or writing in the cache

memory of another process. Cache-based covert and side channels use indirect means and side effects to transfer information from one process to another. One side effect is the variation of cache access delays. The access to a cached memory line is fast, while the access to a previously evicted cache line is slow. This property allows monitoring access patterns, and subsequently leaking information from one process to another.

Covert channels using caches have been known for a long time. Hu [Hu92] in 1992 is the first to consider the use of cache to perform cross-process leakage via covert channels. Two general strategies exist: *prime+probe* [Per05, OST06, TOS10, NS06] and *flush+reload* [GBK11, YF14]. With *prime+probe*, a receiver process fills the cache, then waits for a sender process to evict some cache sets. The receiver process reads data again and determines which sets were evicted. The access to those sets will be slower for the receiver because they need to be reloaded in the cache. With *flush+reload*, a receiver process flushes the cache, then waits for a sender process to reload some cache sets. The receiver process reads data again and determines which sets were reloaded. The access to those sets will be faster for the receiver because they don't need to be reloaded in the cache. The *flush+reload* attack assumes shared lines of cache between the sender and the receiver – and thus shared memory – otherwise the sets reloaded by the sender will not be faster to reload by the receiver than the evicted ones. Indeed, the receiver cannot access sets reloaded by the sender if they don't share memory.

The previous attacks rely on the fact that the sender and the receiver can both target a specific region in the cache (called a *set*). In the absence of any shared memory (which is in general not available across VMs or processes), several conditions create *addressing uncertainty* (term coined in [WXW12]) making it difficult to target a specific set. First, processors implement virtual memory using a Memory Management Unit (MMU) that maps virtual addresses to physical addresses. With virtual machines, hypervisors introduce an additional layer of translation. The guest virtual pages are translated to the guest physical pages, and further to the actual machine pages. The hypervisor is responsible for mapping the guest physical memory to the actual machine memory. A process knowing a virtual address in its virtual machine has no way of learning the corresponding physical address of the guest, nor the actual machine address. Finally, modern processors map an address to a slice using the so-called *complex addressing* scheme which is undocumented.

In Chapter 5, I describe several contributions and a covert channel allowing to pass information from one core to another and to tackle the *addressing uncertainty* without any shared memory.

2.4 Outline of the thesis

This document presents several contribution around content placement, load-balancing and caching policies. It is organized around two axes: (i) system design and (ii) security.

System design

In Chapter 3 and Chapter 4, I present two contributions around the design of content placement, load-balancing and caching policies in distributed systems. While both contributions deal with video content distribution for Video-On-Demand (VoD) systems, they build upon different parts of the network infrastructure. One relies on the edge of the Internet, ie. on home gateways, the other relies on the cloud. The first one - Push-to-Peer - in Chapter 3 is a P2P system where content is proactively pushed to the edge of the Internet on home network gateways. The second one in Chapter 4 is a targeted ad-inlaying system that inserts personalized ads into VoD content. Eventhough the content is personalized some users might share the personalization. Therefore the system must decide which (targeted) content to cache in the cloud instead of recomupting it each time a user accesses it.

Security

In Chapter 5, I present contributions around security applications and security attacks that exploit caches. The first contribution, in Section 5.1, is a trusted timestamping service relying on the caches of Domain Name System of the Internet. Trusted timestamping provides a proof that certain data existed at a certain time. Such a service is indispensable in many situations of the digital world, e.g., online auctions and transaction to ensure correct order of bids and transactions, publication systems to prove that a document was published at a given time. The second contribution, in Section 5.2, is a covert channel that is rendered possible by CPU caches. The covert channel allows two virtual machines in a cloud environment to communicate despite isolation provided by the hypervisor. In this latter contribution, the performance optimization provided by the cache is an enabler for attacks.

The document concludes with an outlook and some future work in Chapter 6.

3

Placement and Load-balancing Policies on Edge Hosts

In this Chapter, we consider the design and analysis of *Push-to-Peer*, a Video-on-Demand (VoD) system in a network of long-lived peers where upstream bandwidth and peer storage are the primary limiting resources. In such a system, video is first pushed to a population of peers. This first step is performed under provider or content-owner control, and can be performed during times of low network utilization (e.g., early morning). Note that as a result of this push phase, a peer may store content in which it itself has no interest, unlike traditional pull-only peer-to-peer systems. Following the push phase, peers seeking specific content will then pull content of interest from other peers. The Push-to-Peer approach is well-suited to cooperative distribution of stored video among set-top boxes or Internet home gateways (that we generally call *boxes*) within a single DSLAM in a DSL network, where the boxes themselves operate under provider control.

We propose several data placement policies and its consequent pull policy for downloading video. We analyze the performance of these policies. Our performance models can be used not only to quantitatively analyze system performance but also to dimension systems so that a given level of user performance is realized.

3.1 Network Setting and Push-to-Peer Operation

The Push-to-Peer system is composed of a content server, a control server, and boxes. The boxes are Set-Top-Boxes (STBs) or Internet home gateways, located at the edge of the network in the users homes. We assume that boxes have hard-disks that can store content that is pushed. The content server pushes content to the boxes during the *push* phase, as described below. A control server provides a directory service to boxes in addition to management and control functionalities.

Content distribution proceeds in two phases in our Push-to-Peer system.

- **Push Phase** During the push phase, the content server pushes content to each of the boxes. We envision this happening periodically, when bandwidth is plentiful (e.g., in the early AM hours), or in background, low priority mode. After pushing content to the peers, the content server then disconnects (i.e., does not provide additional content), until the next push phase. A crucial issue for the push phase is that of data placement: what portions of which videos should be placed on which boxes; we address this problem in Section 3.2.

- **Pull Phase** In the pull phase, boxes respond to user commands to play content. Since a box will typically not have all of the needed content at the end of the push phase, it will need to retrieve missing content from its peers.

Each movie is chopped into *windows* of contiguous data. A full window needs to be available to the user before it can be played back. However a user can play such a window once it is available, without waiting for subsequent data. Windows are typically further divided into smaller data blocks that are stored onto distinct boxes.

We make the some assumptions about the network, and the boxes at the user premises. First, we assume that the upstream bandwidth from each box to the DSLAM is a constrained resource, possibly smaller than the video encoding/playback rate. We will consider the cases that the available upstream bandwidth is either fixed, or can vary over time. In our models a peer that is uploading video to N different peers, equally shares the upstream bandwidth among those N peers. We assume that the downstream bandwidth is large enough so that it is never the bottleneck when a peer is downloading video from other peers (instead, the upstream bandwidths at those other peers are collectively the limiting resource). We thus also assume that the downstream

bandwidth is larger than the video encoding/playback rate. Finally, we assume homogeneous peers, i.e., that all peers have the same upstream link bandwidth and the same amount of hard disk storage.

3.2 Data placement and pull policies

In this Section, we propose the full-striping data placement and code-based data placement schemes. Note that, we don't consider the full striping to be a practical scheme, since it is not resilient to box failures. We present it to obtain the benchmark performance bound of the Push-to-Peer system, which is meant to be compared to the performance of the code-based scheme.

Full Striping scheme

The full striping scheme stripes each window of a movie to all M boxes participating in the Push-to-Peer system. Specifically, every window of size W is divided into M blocks each of size W/M and each block is pushed to only one box *proactively*. Consequently, each box stores a *distinct* block of a window. A full copy of a given window is reconstructed at a particular box by concurrently downloading $M-1$ distinct blocks for the window from the other $M-1$ boxes. Hence a single movie download request generates $M-1$ sub-requests, each targeted at a particular box.

A box serves admitted sub-requests according to the Processor Sharing (PS) policy, forwarding its blocks of the requested video to requesting boxes. PS is an adequate model of fair sharing between concurrent TCP connections, when the bottleneck is indeed the upstream bandwidth.

We further impose a limit on the number of sub-requests that a box can serve simultaneously. Specifically, to be able to retrieve the video at a rate of R_{enc}, one should receive blocks from each of the $M-1$ target boxes at rate at least R_{enc}/M. Hence we should limit the number of concurrent sub-requests on each box to at most $K_{max} := \lfloor B_{up}M/R_{enc} \rfloor$, where B_{up} is the upstream bandwidth of each box. We envision two approaches to handle new video download requests that are blocked because one of the $M-1$ required boxes is already serving K_{max} distinct sub-requests. In the *blocking model*, we simply drop the new request. In the *waiting model*, each of the $M-1$ sub-requests generated by the new request is managed independently at each target box. If the number of concurrent jobs at the target box is below K_{max}, then the sub-request enters service directly. Otherwise, it is put in a FIFO queue that is local to the serving box, and waits there till it can start service.

Code-based placement

We now describe a modification of full striping, namely code-based placement, under which the maximum number of simultaneous connections that a box can serve is bounded by y, for some $y < M - 1$. This scheme applies rateless coding [Lub02, MM03a]. that can generate an infinite number of so-called coded symbols by combining the k source symbols of the original content. These k source symbols can be reconstructed with high probability from any set of $(1 + \epsilon) * k$ distinct coded symbols. In practice, the overhead parameter ϵ falls in $[0.03, 0.05]$, depending on the code that we use [BLM02, MM03a].

The coding scheme we propose consists of dividing each window into k source symbols (the source symbols being the blocks of a window), and generating $C * k = (M * (1 + \epsilon)/(y + 1)) * k$ coded symbols. We call C the expansion ratio, where $C > 1$. For each window, the $C * k$ symbols are evenly distributed to all M boxes such that each box keeps $C * k/M = (1 + \epsilon) * k/(y + 1)$ distinct symbols. A viewer can reconstruct a window of a movie by concurrently downloading any $C * k * y/M$ distinct symbols from an arbitrary set of y boxes out of $(M - 1)$ boxes.

We now define the pull strategy used for the coding scheme. The maximum number, K'_{max}, of sub-requests that can be processed concurrently on each box to ensure delay-free playback now reads $K'_{max} = \lfloor (y+1)B_{up}/R_{enc} \rfloor$. Under the blocking model, a new request is dropped, unless there are y boxes currently handling less than K'_{max} sub-requests. In that case, the new request creates y sub-requests, that directly enter service at the y boxes currently handling the smallest number of jobs. Under the waiting model, each box has a queue from which it selects sub-requests that will be treated by the box. Each new movie download request generates $M - 1$ sub-requests, sent to all other boxes. At a receiving box, such a sub-request either enters service directly, if there are less than K'_{max} sub-requests currently served by that box, or otherwise is stored in a FIFO queue specific to the box. Once a total of y sub-requests have entered service, all the other $M - 1 - y$ sub-requests are killed. Thus each request eventually generates only y sub-requests.

Performance models

In our paper [**SDK$^+$07**], we propose performance models for both the *waiting model* and the *blocking model*. The blocking model predicts the blocking probability of the system given a certain movie request arrival rate. The waiting

model approximates the waiting time of a request. In our paper, we validated these models using simulation and compare the performance of the two placement schemes.

In this document we only report the results of the *waiting model* for full striping. The waiting time D distribution of an arbitrary job for the full striping scheme is:

$$\mathbf{P}(D > t) \approx e^{-(1-\rho)[K_{max}+2t\overline{\sigma}/\overline{\sigma^2}]}. \tag{3.1}$$

We denote by $\overline{\sigma}$ the average job service time. We denote by ρ the normalized load of the system. The details and validation of the model can be read in our paper [SDK$^+$06]. In Section 3.4, we detail how to use the above equation to dimension video prefixes that are pushed to the boxes.

3.3 Optimality properties

In this Section, we demonstrate optimality properties of the full striping and coding schemes, in terms of the demands they can accommodate for stochastic models of demands. We assume that requests for movie j occur at the instants of a Poisson process with rate ν_j. Each such request originates from box m with probability $1/M$, for all $m \in \{1, \ldots, M\}$.

Full striping

Denote by L_j the size of movie j, and by $A_{j,m}$ the amount of memory dedicated to movie j on box m. Then the average size of a download request for movie j is $L_j - (1/M)\sum_{m=1}^{M} A_{j,m}$.

We assume that a single copy of each movie is stored in the system, which can be translated into the constraint $\sum_{m=1}^{M} A_{j,m} = L_j$. It is natural to ask whether under such constraints, there exists a placement strategy that is optimal with respect to the demand rates ν_j that it can accommodate. The following shows that full striping is such an optimal placement strategy:

Proposition 1. *Assume that a single copy of each movie is stored in the whole system. Then under full striping data placement, and for the waiting model above described,*

the system is stable (i.e., download times do not increase unboundedly) whenever the Poisson arrival rates ν_j verify

$$\sum_{j=1}^{J} \nu_j L_j \left(1 - 1/M\right) < M * B_{up}. \tag{3.2}$$

Moreover, for any other placement strategy specified by the $A_{j,m}$, the set of demand rates ν_j that can be accommodated without rejection is strictly smaller than that under full striping.

Proof. Note that for any placement policy in which movies are stored only once, the work arrival rate at a given box m is given by

$$\rho(m) := (1 - 1/M) \sum_{j=1}^{J} \nu_j A_{j,m}. \tag{3.3}$$

Under full striping, one has $A_{j,m} = L_j/M$. Thus condition (3.2) is equivalent to the condition that the work arrival rate $\rho(m)$ is less than the service rate B_{up} of box m. This condition does not depend on m, and is thus necessary and sufficient for stability of the whole system.

Consider now a different placement strategy, for which there exists a pair (j^*, m^*) such that $A_{j^*,m^*} > L_{j^*}/M$. For any demand rates ν_j, $j = 1, \ldots, J$, assume that there exists a pull strategy that can stabilize the system under such demand. Then necessarily, for all $m \in \{1, \ldots, M\}$, one has $\rho(m) < B_{up}$. Summing these inequalities one obtains (3.2), hence such demand can also be handled under full striping.

Consider now a particular demand vector where $\nu_j = 0$ for all $j \neq j^*$, and

$$\nu_{j^*}(1 - 1/M)L_{j^*} = MB_{up} - \epsilon,$$

for some small $\epsilon > 0$. Clearly this verifies (3.2). However, the load placed on box m^* is precisely

$$\rho(m^*) = (1 - 1/M)\nu_{j^*} A_{j^*,m^*}.$$

By our choice of (j^*, m^*), we thus have that

$$\rho(m^*) > (1 - 1/M)\nu_{j^*} L_{j^*}/M.$$

Thus for small enough ϵ, one must have $\rho(m^*) > B_{up}$. Therefore, this box is in overload and the system cannot cope with such demands, while full striping can. □

Coding scheme

We assume additional storage is used per movie as described before. Specifically, we assume that a total storage capacity of $C * L_j$ is devoted to movie j, where $C = M * (1 + \epsilon)/(y + 1)$ is the expansion ratio introduced in the previous section. The solution based on encoding assumes that for movie j, a total quantity of $A_{j,m} \equiv C * L_j/M$ data is stored on each individual box m. This data consists of symbols, such that for any collection of $y + 1 = M/C$ boxes, each movie can be reconstructed from the joint collections of symbols from all these $y + 1$ boxes. We then have the following proposition:

Proposition 2. *By using the pull strategy described in Section 3.2, the system is stable whenever the Poisson arrival rates ν_j verify*

$$\sum_{j=1}^{J} \nu_j L_j \left[1 - C/((1+\epsilon)M)\right] < M * B_{up}. \tag{3.4}$$

Moreover, any scheme that uses $C * L_j$ storage for movie j cannot cope with demand rates ν_j, unless the following ondition

$$\sum_{j=1}^{J} \nu_j L_j \left[1 - C/M\right] < M * B_{up} \tag{3.5}$$

holds.

We omit the proof, which appears in the technical report [SDK+06]. We only note that the average amount of data that needs to be downloaded for a request for movie j is $L_j(1 - C/M)$ when the overall storage devoted to movie j is CL_j, and hence the left-hand side of (3.5) is indeed the rate at which work enters the system, while the right-hand side is an upper bound on the service capacity of the system. Thus with the assumed total storage per movie, Condition (3.5) is indeed necessary to ensure the existence of a pull strategy for which the system is stable. The coding scheme is indeed nearly optimal, since Conditions (3.5) and (3.4) coincide when the overhead parameter ϵ tends to zero.

3.4 Sizing prefixes

We now show how to further optimize content placement assuming extra storage is available. We assume there are J movies, all encoded at a constant bit rate R_{enc}, and denote by L_j the size of movie j.

For movie j, we assume that a prefix of size P_j is stored locally on each box. This ensures that each user can play back the first $t_j := P_j/R_{enc}$ seconds of movie j without downloading extra content. We further assume that encoded symbols are created and placed on each box so that for each movie j, its remainder can be reconstructed from the symbols present at any $y+1$ boxes.

Let S denote the memory space available on each box. The above described placement strategy will be feasible provided the following constraint is satisfied:

$$\sum_{j=1}^{J} P_j + (L_j - P_j)/(y+1) \leq S. \tag{3.6}$$

Denote by ν_j the rate of requests for movie j. The amount of movie j that needs to be downloaded for the playback is then

$$\sigma_j = \frac{y}{y+1}(L_j - P_j). \tag{3.7}$$

Indeed, the prefix of size P_j is stored locally, as well as a fraction $1/(y+1)$ of the remainder of the movie. The normalised load on the system is thus:

$$\rho = \frac{\sum_{j=1}^{J} \nu_j \sigma_j}{B_{total}}. \tag{3.8}$$

$B_{total} = MB_{up}$ is the total service capacity, *i.e.*, the aggregated upstream bandwidth.

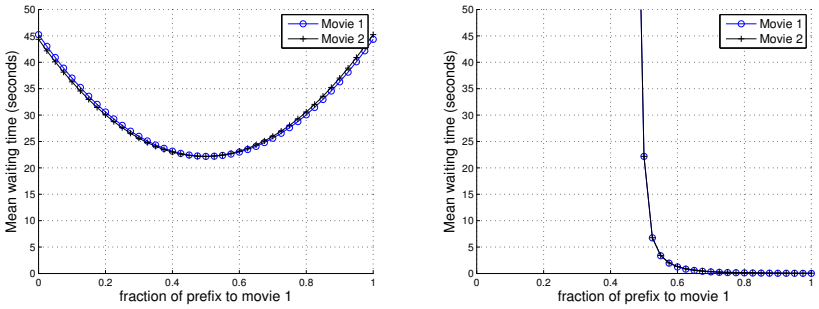
The evaluations (3.1) give us an approximation of the distribution of the delay D between request initiation and download beginning. Using locally stored movie prefixes, the playout delay - as experienced by the end-user - can be reduced because playback can start t_j seconds before download starts. We obtain the following formula for the average delay $\bar{D}_j$ experienced by requests for movie j:

$$\begin{aligned}\bar{D}_j &= E\left[\max(0, D - t_j)\right] \\ &= \int_{t_j}^{\infty} (x - t_j) \frac{2\overline{\sigma}(1-\rho)}{\overline{\sigma^2}} e^{-(1-\rho)[K_{max} + 2x\overline{\sigma}/\overline{\sigma^2}]} dx.\end{aligned}$$

We thus obtain the formula

$$\bar{D}_j = \frac{\overline{\sigma^2}}{2\overline{\sigma}(1-\rho)} e^{-(1-\rho)[K_{max} + 2t_j\overline{\sigma}/\overline{\sigma^2}]}. \tag{3.9}$$

We use a simple example to illustrate how a fixed amount of memory in a box can be optimally allocated to preload prefixes of movies depending on their relative popularities. Figures 3.1a and 3.1b show plots of the mean waiting times $\overline{D}_j$ obtained from Formula (3.9). In each case, there are two



(a) (b)

Figure 3.1: Waiting time against prefixes with (a) balanced and (b) distinct popularity. (a) $\nu_1 = \nu_2 = 0.99$, $R_{enc} = B_{total} = 1$, $L_1 = L_2 = 1$, $P_1 + P_2 = 1$, $y \gg 1$ (b) $\nu_1 = 0.99 * 4/3$, $\nu_2 = 0.99 * 2/3$, $R_{enc} = B_{total} = 1$, $L_1 = L_2 = 1$, $P_1 + P_2 = 1$, $y \gg 1$

movies, and there is a fixed amount of memory that can be used for prefixes of either or both movies. In Figure 3.1a, the popularities of both movies are same. In this case, the figure indicates that both movies should get prefixes of equal sizes. Note that for equal popularities, varying prefixes does not change the normalized load ρ. Also, it does not affect the average service time $\bar{\sigma}$. It would appear then that one movie would benefit from having a larger prefix. This is however not the case, because unbalanced prefixes lead to a large variance in the service times and thus a large second moment $\overline{\sigma^2}$.

In Figure 3.1b, movie 1 is twice as popular as movie 2. The figure indicates that it is beneficial to both movies to allocate the prefix memory to movie 1. Here, by storing large prefixes for movie 1, we reduce the system load ρ, and this is the leading effect.

3.5 Randomized Job Placement

In the previous Section, we restricted ourselves to the case where all boxes are centrally coordinated. With such an assumption the job placement strategies, i.e. the decision where to place and serve the sub-requests of a job, are optimal. However, in practice a centralized system does not scale. In this Section, we therefore propose distributed load balancing strategies for the selection of serving peers.

The strategy we consider for initial job placement is as follows. We assume that original content placement has been done according to the coding strategy. When a download request is generated, d distinct boxes are randomly chosen from the overall collection of M boxes. The load, measured in terms of fair bandwidth share that a new job would get, is measured on all probed boxes. Finally, sub-requests are placed on the y least loaded boxes among the d probed boxes, provided that each of the y sub-requests gets a sufficiently large fair bandwidth share, i.e. larger than or equal to $R_{enc}/(y+1)$. If any of the least loaded boxes cannot guarantee such a fair share, then the request is dropped.

In the paper [**SDK$^+$07**], we determine fixed point equations that characterize the fraction of boxes holding a given number of sub-jobs in equilibrium. The fixed point equations allows us to calculate (using numerical approximation) the parameters p_j - being the fraction of boxes serving j sub-jobs. Using the parameters p_j, we can calculate the rejection probability according to the formula

$$p_{reject} = \sum_{i=d-y+1}^{d} \binom{d}{i} p_{K_{max}}^{i} (1 - p_{K_{max}})^{d-i}.$$

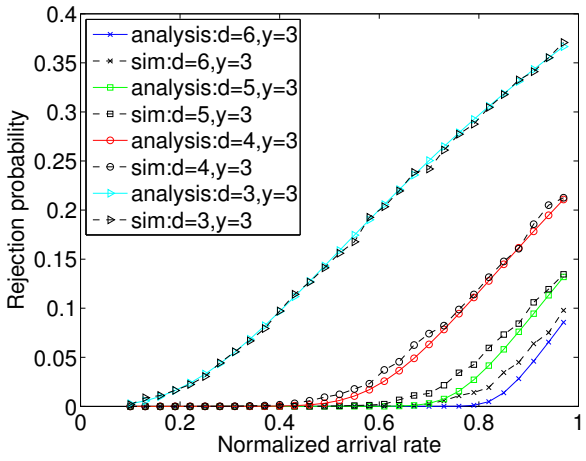


Figure 3.2: Numerical solutions and simulation results for rejection probability using the proposed load balancing scheme

We validate our model by simulations. Figure 3.2 shows the numerical solutions and simulation results we obtain for distinct choices of parameters (y, d) for varying normalized load, and setting K_{max} to 3. We observe that even

at normalized loads close to 100% (service rate and arrival rates are equal), the rejection probabilities remain small: below 15% when only two additional boxes are probed and down to 10% when three additional boxes are probed.

3.6 Conclusion

We proposed Push-to-Peer, a peer-to-peer approach to cooperatively stream video using push and on-demand pull of video contents. We showed the theoretical upper performance bounds that are achieved if all resources of all peers are perfectly pooled, and present the placement (namely full-striping and coding scheme) and pull policies that achieve those bounds. However, perfect pooling is only possible with global knowledge of system state, which in practice is not feasible. Therefore, we proposed and analyzed a randomized job placement algorithm. We also studied how to store movie prefixes on boxes, to further optimize the system usage.

Since our initial proposal, the Internet and its usages largely evolved. An important part of the video consumption moved to mobile terminals such as tablets or smartphones. The video is not necessarily consumed at home anymore. With these changes many of our assumptions do not hold anymore, e.g., the edge-devices are not necessarily pooled around a common DSLAM and the bandwidth capacity and availability for the mobile terminal may fluctuate. This is an opportunity to revisit our approach. Is it possible to prefetch and cache locally on the mobile terminals? Can we still rely on a home gateway infrastructure, that would prefetch and cache video content and serve it to the moving mobile terminals?

Finally, the peer-to-peer paradigm lost traction these last years and today most of the commercially deployed services rely on the cloud or on Content Delivery Networks (CDNs). Still, these architectures may also benefit from policies that pro-actively push some content to a particular server or location. Similar approaches to the one presented in this Chapter may therefore be cast and studied for these architectures.

4

Caching Policies in the Cloud

The research community moved from the peer-to-peer paradigm to more centralized solutions, namely the Cloud or Content-Delivery-Networks. This trend was backed by the commercially deployed services that require high service guarantees - which are easier to achieve in centralized and controlled environments. Even if more centralized, these architectures are still distributed in nature and need algorithms and policies to place and distribute content. In this Chapter, we present a particular instance of cloud-based caching policies.

Cloud-based caching consists in storing frequently accessed files on cloud storage services (e.g., Amazon S3); it uses cloud storage as a data cache. Such a cache is beneficial for files that would have been recomputed using cloud compute or retrieved from a private data center at each access without a cache.

There are two main differences between cloud-based caching and classical network caching. First, cloud-based caching does not impose a limit on the cache capacity; the size of the cache is virtually infinite. Second, cloud-based caching adopts a pay-per-use cost model. Generally, the cost is proportional to the volume of cached data times the time data is cached. There is also a cost of not caching an item, e.g., the cost of (re)computing an item using cloud compute or the cost of fetching the item from a remote location.

The above differences impact the design of *cache policies* which determine how to efficiently use cache resources. Classical caches implement *cache replacement algorithms*, such as Least Recently Used (LRU) or Least Frequently Used (LFU), that decide which item to evict from the cache and to replace with a new item. These algorithms try to maximize the hit ratio of the cache.

With cloud-based caches it is not necessary to systematically *replace* an item, since the cache capacity is neither fixed nor bound. Further, the objective of the cache policy is not necessarily to maximize the hit ratio.

In this Chapter, we design and evaluate cache policies with the objective of minimizing the cost of a cloud-based system. At each access of an item, the item is either recomputed at the cost of cloud compute or stored at the cost of cloud storage. We focus on time-based cache policies. Time-based cache policies calculate how long an item should stay in the cache irrespective of a cache size limit. We show that these policies are easier to operate than classical size-based policies managing a fixed capacity. Time-based cache policies also scale with the demand and outperform size-based policies.

4.1 Motivating Example

We instantiate and motivate the above caching problem with a cloud-based personalized video generation and delivery platform; in this use-case, video sequences contain pre-defined ad-placeholders, typically billboards that are included in the video content. Ads are dynamically chosen according to the end-user's profile and inlayed into the appropriate placeholders of the video using cloud compute. The video is cut into chunks such that the video processing only occurs on relevant portions. Of course, several users (with overlapping profiles) may be targeted with the same ad for the same movie. Instead of recomputing these personalized chunks for each user (called items in the rest of the paper), the platform can cache them using cloud storage.

During content distribution, we use cloud compute to generate (on-the-fly) targeted chunks, called hereafter items. Generating an item consists in (i) decoding the corresponding chunk under the mezzanine format, (ii) inserting the ad in every relevant frame of the video sequence using the corresponding metadata, (iii) compress the resulted video sequence under a distribution format compatible with the delivery network, and (iv) serve it to the user and store the item in a cache if needed. When a user asks for a content, the system decides which ads to insert into each placeholder of that content for that particular user. The generation of the item using cloud compute can be economized if the item is present in the cache.

4.2 Cache policies for cloud systems

In this section, we suppose that requests for an individual item k (corresponding to one chunk in a movie i with an ad j) arrives according to an homoge-

neous Poisson process of intensity λ_k. We study the cost for serving a request when a given item is stored in a cache. The item is deleted from the cache if it is not accessed for more than T_k seconds. If the item is not available from the cache it is computed. The storage cost is S dollars per item per second, and the computation cost is C dollars per item per computation. Note that, the cost of transmission and access counts can be included into costs C and S.

Let t be a continuous variable describing the time since the last access to the item k. Since the request arrivals follow an homogeneous Poisson process, the probability that the next request for the item k arrives at time t is

$$p(t) = \lambda_k \exp^{-\lambda_k t} \tag{4.1}$$

Let X_k be a continuous random variable describing the cost for serving an item k. For a given request, if $t < T_k$, then the item is served from the cache and there are only storage costs $X_k = tS$. If $t > T_k$, then the item is stored for T_k seconds, and re-computed when accessed. Hence, the expected cost for serving the item k is

$$\mathbb{E}[X_k] = \int_0^{T_k} p(t) t S \, \mathrm{d}t + \int_{T_k}^{\infty} p(t)(T_k S + C) \, \mathrm{d}t \tag{4.2}$$

which simplifies to

$$\mathbb{E}[X_k] = \frac{S}{\lambda_k} + \frac{(\lambda_k C - S) \exp^{-\lambda_k T_k}}{\lambda_k} \tag{4.3}$$

The expected cost $\mathbb{E}[X_k]$ has a minimum for $T_k = 0$ if $\lambda_k < \frac{S}{C}$, or a minimum for $T_k = \infty$ if $\lambda_k > \frac{S}{C}$.

Assuming that we perfectly know λ_k, (4.3) allows us to determine an ideal caching policy: *(i)* never cache the item k if its arrival rate λ_k is smaller than the ratio cloud storage cost over cloud compute cost ($\lambda_k < \frac{S}{C}$); *(ii)* indefinitely cache the item k if its arrival rate is greater than the ratio cloud storage cost over cloud compute cost ($\lambda_k > \frac{S}{C}$).

In practice λ_k is not known. Various policies, described in th next Sections, can be derived from these observations. The differences of these policies lie in the a-priori knowledge that we suppose available:

- We only know general laws governing the popularity distribution of movies, e.g. following a Zipf distribution.
- We estimate the arrival rate for each item i by counting the number of requests occurring over a sliding temporal window.
- We assume that when an event occurs at time t, we know at which time $t' > t$ the item will be requested next (*i.e.*, we can predict the future).

No knowledge of λ_k, Global TTL

We assume that requests are distributed across movies according to a Zipf distribution of exponent s_m on N_m movies [FRRS12, CKR+07]. The probability that a request is for a movie i is

$$p(M = i) = \frac{1}{i^{s_m}} \frac{1}{H_{N_m,s_m}} \tag{4.4}$$

where $H_{N,s} = \sum_{i=1}^{N} \frac{1}{i^s}$ is the generalized harmonic number.

We also assume that ads are distributed across movies according to a Zipf distribution of exponent s_a on N_a ads. N_a, s_a and N_m, s_m denote the Zipf distribution parameters for ads and movies respectively. The probability that a request is for an ad j is

$$p(A = j) = \frac{1}{j^{s_a}} \frac{1}{H_{N_a,s_a}} \tag{4.5}$$

We note $p_{i,j}$ the probability that the i-th movie is served with the j-th ad (*i.e.*, $k = (i, j)$). We consider that the choice of movies and ads are independent; we have

$$p_{i,j} = P(M = i) \cdot P(A = j) = \frac{1}{i^{s_m} H_{N_m,s_m}} \frac{1}{j^{s_a} H_{N_a,s_a}} \tag{4.6}$$

For the sake of clarity, we will note $p_{i,j}$ to designate this quantity in the rest of the paper.

We assume that the requests for each item also follow a homogeneous Poisson process, so that each item i, j is served at a rate $\lambda_{i,j} = \lambda \cdot p_{i,j}$ Hence, if the next request for some item arrives at time t and the next request for each specific item i, j arrives at time $t_{i,j}$, we have

$$P(t_{i,j} = t) = p_{i,j} \tag{4.7}$$

which means that movies and ads are distributed according to the Zipf-laws previously defined.

Let X be the cost for serving any item. We do not distinguish items and set a global timeout T that apply to all items (*i.e.*, $T_{i,j} = T$). In this case, the expected cost X for serving an item is

$$\mathbb{E}[X] = \sum_{i=1}^{N_m} \sum_{j=1}^{N_a} p_{i,j} \left(\frac{S}{\lambda_{i,j}} + \frac{(\lambda_{i,j} C - S) \exp^{-\lambda_{i,j} T}}{\lambda_{i,j}} \right) \tag{4.8}$$

$$\mathbb{E}[X] = \sum_{i=1}^{N_m} \sum_{j=1}^{N_a} \frac{S(1 - \exp^{-p_{i,j}\lambda T})}{\lambda} + p_{i,j} \cdot C \cdot \exp^{-p_{i,j}\lambda T} \tag{4.9}$$

This policy is simple to implement yet fails at distinguishing items, and requires that the distribution of requests among movies matches well-known Zipf law. In the following, we design a policy that adapts the strategy to each item popularity rather than having a single strategy for all items.

Estimation of λ_k, Individual TTL

In this policy, the arrival rate for each item i is estimated by counting the number of requests occurring over a sliding temporal window. Following the previous observation that items for which $\lambda_{i,j} < \frac{S}{C}$ should not be stored at all, and that items for which $\lambda_{i,j} > \frac{S}{C}$ should be stored indefinitely, this policy compares the estimate of $\lambda_{i,j}$ to $\frac{S}{C}$ to choose between storing or not. The decision of storing or not is continuously revisited each time the observed $\lambda_{i,j}$ changes.

In this case, the expected cost X for serving an item is such that $T_{i,j} = 0$ for all items such that $\lambda_{i,j} < \frac{S}{C}$ and $T_{i,j} = \infty$ for all items such that $\lambda_{i,j} > \frac{S}{C}$. If we assume that $\lambda_{i,j}$ is perfectly estimated and that movies and ads are distributed according to Zipf-laws as described in the previous section, then the expected cost X for serving an item is

$$\mathbb{E}[X] = \sum_{i=1}^{N_m} \sum_{j=1}^{N_a} p_{i,j} \left(\frac{S}{\lambda_{i,j}} + \frac{(\lambda_{i,j}C - S) \exp^{-\lambda_{i,j}T_{i,j}}}{\lambda_{i,j}} \right) \tag{4.10}$$

which can be rewritten as

$$\mathbb{E}[X] = \sum_{i=1}^{N_m} \sum_{j=1}^{N_a} p_{i,j} \left(\frac{S}{\lambda_{i,j}} (1 - \exp^{-\lambda_{i,j}T_{i,j}}) + C \exp^{-\lambda_{i,j}T_{i,j}} \right) \tag{4.11}$$

Leveraging the fact that $T_{i,j} = 0$ for all items such that $\lambda_{i,j} < \frac{S}{C}$ and $T_{i,j} = \infty$ for all items such that $\lambda_{i,j} > \frac{S}{C}$, we obtain

$$\mathbb{E}[X] = \sum_{i=1}^{N_m} \sum_{j=1}^{N_a} p_{i,j} \left(\mathbf{1}_{\lambda_{i,j} \geq \frac{S}{C}} \cdot \frac{S}{\lambda_{i,j}} + \mathbf{1}_{\lambda_{i,j} < \frac{S}{C}} \cdot C \right) \tag{4.12}$$

These formulas provide the cost of the best policy achievable if we have a perfect knowledge of $\lambda_{i,j}$ (and not an estimate). This allows comparing the performance of the scheme we implement, which estimates $\lambda_{i,j}$, against a policy with a perfect knowledge of $\lambda_{i,j}$.

Perfect knowledge of event occurrences, Lower bound

Finally, this is an ideal policy which provides a lower bound that is not attainable: this policy assumes that when an event occurs at time t, we know at which time $t' > t$ the item will be requested next (*i.e.*, we can predict the future).

We first determine the expected cost for serving a given item k.

$$\mathbb{E}[X_k] = \int_0^{C/S} p(t)St \, \mathrm{d}t + \int_{C/S}^{\infty} p(t)C \, \mathrm{d}t \tag{4.13}$$

$$\mathbb{E}[X_k] = \frac{S}{\lambda}\left(1 - \exp^{-\frac{C\lambda_k}{S}}\right) \tag{4.14}$$

If we assume that movies and ads are distributed according to a Zipf-laws as described previously, then the expected cost X for serving an item $k = (i, j)$ is

$$\mathbb{E}[X] = \sum_{i=1}^{N_m}\sum_{j=1}^{N_a} p_{i,j}\frac{S}{\lambda}\left(1 - \exp^{-\frac{C\lambda_{i,j}}{S}}\right) \tag{4.15}$$

An interesting feature of these three time-based caching policies is that their modeling and analysis is simpler than traditional capacity-based caching policies. Items do not interact with each other and in particular are not evicted from the cache so that another item can be stored. However, such time-based policies are adapted only to platforms that provide variable and unbounded amount of resources. Such platforms are now available from cloud provider that have a pay-per-use model with potentially infinite amount of resources.

4.3 Evaluation

We implemented a discrete-event simulator of a cloud-based caching system. The simulator models the arrivals of user requests, movie and ad popularities, and calculates the cloud system cost per item, given some cache policy. We use both synthetic and trace-based simulations.

Synthetic simulations use a Poisson arrival process to model request arrivals and two Zipf distributions to model movie and ad popularities. At each user request arrival, the simulator selects a movie i according to a first Zipf distribution and an ad j according to a second Zipf distribution. The tuple (i, j) corresponds to an item as described in Section 4.2. We use the values

$s_m = 0.8$ and $N_m = 10,000$ [FRRS12, CKR+07] for the movie Zipf distribution, and $s_a = 0.94$ and $N_a = 5,000$ for the ad Zipf distribution.[1] We consider different global arrival rates $\lambda = 10$, $\lambda = 100$, and $\lambda = 300$ requests per hour.

Trace-based simulations rely on three different traces, representing three different types of Video-On-Demand systems: *(i)* the Netflix prize dataset [BL07], representing a large premium content Video-On-Demand system, *(ii)* Youtube traces [CKR+07], representing a large user-generated content video system and *(iii)* Daum traces [CKR+07], representing a small user-generated content video system. The traces provide the simulator with user requests at various times: a user request corresponds to a movie view with Youtube and Daum or a movie rating with Netflix. As with synthetic simulations, for each user request the simulator selects an ad according to a Zipf distribution with $s_a = 0.94$ and $N_a = 5,000$.

We derive our cost model from the cost model of Amazon AWS. We consider videos with an video output bitrate of 3.5Mbit/s, which provides 720p HD quality. The video is split into chunks of length of 10 seconds. Our measurements indicate that we can compute an item in real-time using Amazon EC2 M1 Large instances. At the time of writing these instances cost 0.26\$ per hour for Europe. We can derive a cloud compute cost of $C = 7.2 \times 10^{-4}$ \$ per item. Similarly, using the cloud storage costs of Amazon S3 we can compute a storage cost of $S = 4.86 \times 10^{-7}$ \$ per hour per item assuming a cost of 0.08\$ per gigabyte per month.[2]

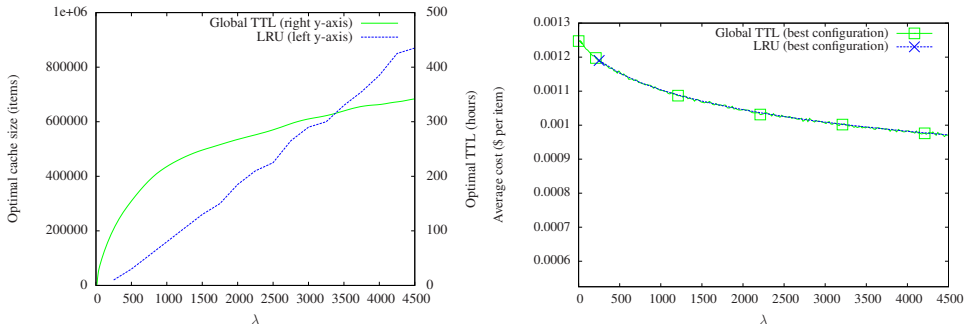
Finally, transmission costs are the same for cached and non-cached chunks, since Amazon only charges for traffic that leaves the Amazon cloud. In our model, the transmission cost accounts for 5.25×10^{-4} \$ per item. In the rest of this section, all plots showing the average cost per item use 5.25×10^{-4} \$ as their baseline for the y-axis.

The simulator implements the following cache policies:

– *Individual TTL:* simulates the policy described in Section 4.2. It continuously adds or removes items to cache depending on an item arrival rate λ_k observed during the sliding window on past item requests. With a $\lambda_k \leq \frac{S}{C}$ the simulator removes the item k from the cache, and with a $\lambda_k \geq \frac{S}{C}$ the simulator adds the item k to the cache.

[1] We extracted the latter parameter from an internal dataset of ad-popularity distribution of ads inserted on web pages.

[2] Changing the quality of the video to 1080p HD or SD would not fundamentally change our results. Our measurements indicate that the ratio S/C does not significantly change from one resolution to another.



(a) Sensitivity to user request arrival rate.

(b) Cost per item comparison. Both policies are parameterized at their best configuration (optimal cache size, and optimal TTL).

Figure 4.1: Comparison of LRU and Global TTL.

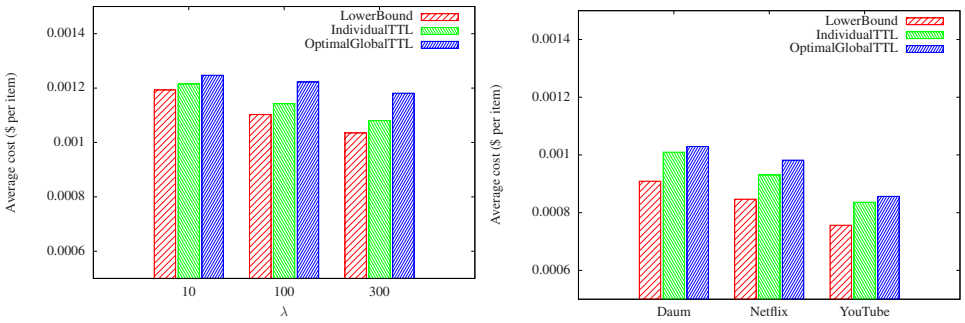
We use a sliding window with a duration of 1,500 hours, which approximately corresponds to the value of $\frac{C}{S}$. We show in ou paper **[SNS14]** that a sliding window duration of $\frac{C}{S}$ provides best performance in most settings.

– *Lower bound:* simulates the policy described in Section 4.2. It decides on each item request whether the item should have been kept in cache since the preceding request for the same item. The policy picks the cheapest of both choices. This computes a lower bound of the system costs for one particular simulator run. There is no cache size limit.

– *LRU:* implements classical LRU policy with a fixed cache size. The caching policy takes as parameter a cache size.

Comparing Size Based and Time Based Cache Policies

We now compare a size based policy, namely *LRU*, and a time based policy, namely *Global TTL*. We consider the cost per item for each policy and the sensitivity of the policy against increasing user request arrival rates. We use the synthetic simulations for the LRU policy and the analytical model for the *Global TTL* policy. For both policies we determine the cache parameter (the size for *LRU*, the TTL for *Global TTL*) that minimizes the average cost per item for a given user arrival rate. It is not our objective to minimize the cache miss



(a) Using Poisson arrivals. (b) Using Daum, Netflix and YouTube traces.

Figure 4.2: Cache policies cost per item.

probability. We use our simulator to compute the optimal *LRU* cache size in function of the user request arrival rate, and our analytical model to compute the optimal *Global TTL* in function of the user request arrival rate.

Figure 4.1a plots the cache parameter that minimizes the cost as a function of the user request arrival rate for both policies. The *LRU* policy is more sensitive to the user request arrival rate than the *Global TTL* policy. The optimal cache size for the *LRU* policy increases as the user request arrival rate increases, while the optimal *Global TTL* seems to be bounded to a value less than 400 hours, as the user request arrival rate increases.

Figure 4.1b shows the minimal cost of both *LRU* and *Global TTL* policies as a function of the user request arrival rate. The cost of both policies perfectly fit. The optimal *Global TTL* provides the same cost savings as the optimal *LRU* policy with less sensitivity to the user request arrival rate. Therefore, we do not consider the *LRU* policy in the remaining of our work.

Time Based Cache Policies Evaluation

We first compare the *Global TTL*, *Individual TTL* and *Lower Bound* policies using our analytical model. For the *Global TTL* policy, we determine the TTL value that minimizes the cost for this policy (the TTL minimizing *Global TTL* is 0, 60 and 120 hours for a λ of 10, 100 and 300 respectively). Figure 4.2a plots the cost per item for each policy. *Individual TTL* systematically performs better than *Global TTL* and approaches the *Lower bound* policy.

We now evaluate the cost per item of *Global TTL*, *Individual TTL* and *Lower bound* policies using traces from Netflix, YouTube and Daum. For the *Global*

TTL policy we determine the TTL value that minimizes the cost for this policy. We approximate this TTL value by computing the global arrival rate for a trace and searching for the TTL value that minimizes Equation (4.9). The optimal TTL values for the *Global TTL* policy are 100, 275 and 300 hours for Daum, Netflix and YouTube respectively.

Figure 4.2b plots the cost per item for the considered traces and policies. Although real traces do not follow exponential laws, the *Individual TTL* policy outperforms the *Global TTL* policy for the three traces. *Individual TTL* policy provides therefore two main advantages : *(i)* it is practical to deploy as *Individual TTL* can be inferred using a sliding window and does not require an a priori knowledge of the user request arrival rate and content and ad popularity; *(ii)* it provides better performances than optimal *Global TTL* or LRU based policies (which require a priori knowledge of system parameters for optimizing the cache size or the TTL).

4.4 Conclusion

We proposed caching policies tailored to the specificities of cloud-based caching, which impose no limit on the cache capacity and adopt a pay-per-use cost model. The proposed caching policies therefore depart from classical caching policies such as LRU that manage a fixed capacity. The objective of the proposed caching policies is to reduce the cloud cost.

We provided analytical models for time-based caching and derived a caching policy, *Individual TTL*, that is easy to operate, since it does not require any a priori knowledge of the user arrival rates or movie popularities. *Individual TTL* minimizes the cost individually for each item, based on the observed item features. The caching policy adapts over time to evolving request rates or prices.

The results presented in this Chapter can apply broadly to systems where there exist a cost for storing and for generating an item on a pay-per-use basis with unbounded resources. As a perspective, it could be interesting to perform a similar work on other aspects of systems that have been studied with the assumption that resources are fixed or bounded, and to reconsider the design decisions that have been taken. This can lead to simpler or more efficient systems.

5

Exploiting Caches

In the previous chapters we built or analyzed caching systems with its primary purpose in mind: reducing the access time to some item or optimizing the cost of a system. As we show in this Chapter, caches can also be exploited and used for different purposes they were not initially designed for. We present two such cases in this Chapter. In Section 5.1, we rely on DNS caches to build a trusted timestamping service. In Section 5.2, we focus on covert channels that are rendered possible by CPU caches. The covert channels allow two virtual machines in a cloud environment to communicate despite isolation provided by the hypervisor. In this case, the performance optimization provided by the cache and its side-effects are an enabler for attacks.

5.1 Short-lived trusted timestamping using DNS cache resolvers

Trusted timestamping, i.e., proof that certain data existed at a certain time, is indispensable in many situations of the digital world. Examples of such situations include: online auctions and transactions to ensure correct order of bids and transactions; electronic voting to ensure that the vote was cast at an allowable time; publication systems to prove that a document was published at a given time. Trusted timestamping generally relies on trusted and centralized timestamping authorities that provide the timestamp. This introduces a single point of trust, a single point of failure and may limit the scalability and availability of the service.

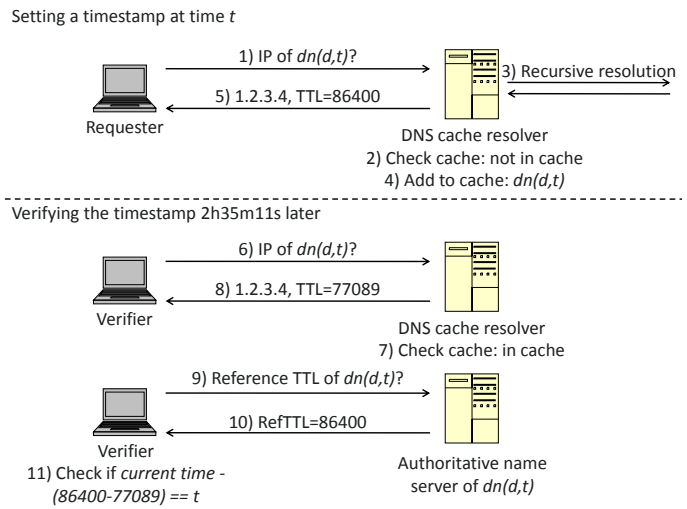


Figure 5.1: Sketch of DNStamp scheme using a single domain name and a single resolver. $dn(d,t)$ denotes a domain name derived from data d to be timestamped and the timestamping time t, using a one-way function.

We propose a completely distributed timestamping scheme, called DNStamp, that takes advantage of the Domain Name System (DNS) caches. In order to timestamp data with DNStamp, a requester sends recursive resolution requests to a list of DNS resolvers. The resolution requests contain domain names, which have been derived from the timestamping time (i.e. the current time) and data to be timestamped using a one-way function. Similarly, the list of resolvers is selected using another one-way function. At reception of above resolution requests, the resolvers add new DNS cache entries of the requested domain names into their cache. The resolvers keep these entries in their cache during a certain Time-To-Live (TTL), a value that is maintained and continuously decremented for each cache entry. The resulting timestamp is short-lived, typically with a validity period of several days - a limit imposed by the TTLs supported by the DNS. In order to verify the timestamp, once data is released, a verifier reiterates the procedure of the requester and retrieves the remaining TTL for each DNS entry. In addition, the verifier asks for the reference TTL at each domain name's authoritative server. The verifier can calculate the time of the timestamp using the current time, the reference TTL and the remaining TTL. Figure 5.1 sketches the DNStamp scheme using a single resolver and a single domain name.

5.1.1 Problem and Objectives

Trusted timestamping consists in proving that certain data d existed at a point in time t_R. More precisely, a generic timestamping process involves three steps: (i) at time t_R, a *requester* timestamps a digest $y = h(d)$ (h is a hash function), which results in the generation of a timestamp T, (ii) at time t_P, $t_P \geq t_R$, the requester publishes the timestamp T (iii) at time t_V, $t_V \geq t_P$, a *verifier* verifies whether T was actually produced at time t_R as a function of digest y. We consider a timestamp to be *trusted* if it resists to *back-dating*, *forward-dating* and *availability* attacks. These notions, the considered adversaries, and a detailed security analysis is described in our paper **[NHO14]**.

5.1.2 Timestamping using the DNS

Requesting a new timestamp

With DNStamp, a requester generates a list of domain names $D = \langle dn_1, \ldots, dn_n \rangle$ and a list of resolvers $R = \langle res_1, \ldots, res_n \rangle$, as described later, which both depend on data digest $y = h(d)$, the generation time t_R and the duration α of the timestamp. Then, she performs the resolution $dn2ip(dn_j, res_j)$ of each domain name $dn_j \in D$ using the resolver $res_j \in R$; she forces recursive resolution in her DNS resolution requests. As an effect, resolvers that did not cache a requested domain name add this domain name to their cache and set the reference TTL ref_i. As specified by the DNS standard, the resolver starts decrementing the TTL and deletes the domain name from its cache when the remaining TTL reaches 0. If an entry dn_j was already cached by the resolver res_j, the domain reference TTL will not be set, and the resolver will continue to decrement the remaining TTL.

The requester generates the above lists D and R as follows. We suppose that a list of valid resolvers, denoted $rlist$, is provided. To compute D, the requester generates a list of m IP addresses $ip_1 \ldots ip_m$ using a hash function h_a such that $ip_i = h_a(y||t_R||\alpha||i)$. $||$ denotes the concatenation. She also generates a list of reverse resolvers using a hash function h_b as follows: $inv_i = rlist[h_b(y, i)]$. She then performs the reverse resolutions $ip2dn(ip_i, inv_i)$ of each ip_i using inv_i. The reverse resolution can fail for some IP addresses, since not all possible IP addresses have a corresponding domain name. The result is the list of domains $D = \langle dn_1, \ldots, dn_n \rangle$ with $n < m$. Let I be the list of indices j such that the reverse resolution of ip_j returns a valid domain name. R is computed such that $res_i = rlist[h_c(y||t_R||\alpha||i)]$ where $i \in I$.

The resulting timestamp is $request(y||t_R||\alpha) = T = (D, I)$.

Publishing the timestamp

At time t_P, the requester publishes T to the world (web) or to a group (social network), or directly to some verifiers (email). We also recommend publishing y and $t_R||\alpha$ along with the timestamp T.

Verifying an existing timestamp

We suppose that the verifier retrieved T, y, t_R and α. The verifier computes the list of resolvers R using the same algorithm as the requester. She then retrieves the remaining TTL for each domain name $dn_j \in D$ using the resolvers $res_j \in R$. In addition, she retrieves the domain reference TTL for each domain name in D by querying the respective authoritative servers (see details in Section 5.1.3). The verification succeeds if, for at least 50% of the domain names of D, the difference between the domain reference TTL (ref_i) and the remaining TTL (rem_i) is consistent with the time provided with the proof, i.e. $t_V = t_R + (ref_i - rem_i)$. Finally, having y, t_R and α, she can also generate the list of domain names D and verify if it is equal to the one contained in T. The above procedure corresponds to checking whether $verif(request(y||t_R||\alpha)) = t_R$.

5.1.3 Measurements and experiments

We have implemented DNStamp by extending and adapting the EphPub command-line tool [CDFK11]. The prototype takes as input the file to be timestamped, the duration the timestamp should stay valid and an option indicating whether to request or verify the timestamp.

Retrieving a timestamp

Figure 5.2 shows the number of cache entries our prototype could successfully read while continuously retrieving a timestamp over 40 hours. The timestamp completely vanishes after 24 hours. This behavior is normal since all remaining TTL reach 0 after 24 hours. About 90% of the cache entries can be read right after the timestamp has been set. The missing 10% correspond to the cache entries that could not be successfully set by the requester. Indeed, some resolvers either refuse a connection (returning a DNS `REFUSED` message) or simply time out. This ratio slightly decreases, and after 20 hours and until the expiration of the timestamp, about 80% of the cache entries are successfully retrieved. The decrease of valid cache entries typically comes from cache leaks

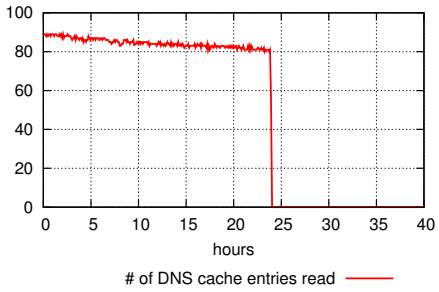
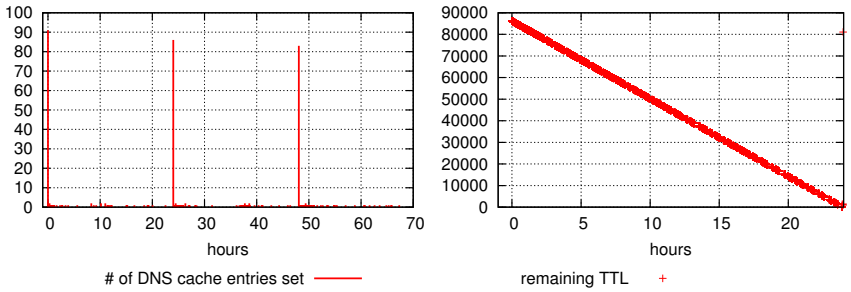


Figure 5.2: Number of cache entries read. Numbers are averages from five 1 day timestamps T continuously verified during 40 hours.



(a) (b)

Figure 5.3: (a) Cache entries successfully set to the reference TTL during 70 hours continuous overwriting of an existing timestamp. (b) Remaining TTL representing 98% of measured cache entries during the validity period (24 hours) of the initial timestamp.

[KPN+93], *i.e.*, DNS resolvers that decide to evict some cache entries before the TTL expires. DNStamp is designed to support failures and the remaining fraction of valid cache entries is sufficient to validate the timestamping time.

Overwriting an existing timestamp

We now experimentally demonstrate that an adversary with no particular privileges is not capable of overwriting the remaining TTL of an existing timestamp. Other set of adversaries and attacks are discussed in the paper [**NHO14**]. We conduct the following experiment to demonstrate this property. We first

generate a valid 1 day timestamp consisting of 100 cache entries. Then, we continuously try to overwrite the existing timestamp during 70 hours. We verify whether we can set cache entries to the reference TTL and monitor the remaining TTL each time we try to overwrite a cache entry.

Figure 5.3a shows the number of cache entries that could be successfully set to the reference TTL. At $t = 0$ and at multiples of 24 hours between 80 and 90 cache entries could be updated. This is normal, since this corresponds to the initial timestamp request (at $t = 0$) and the subsequent expirations of the timestamp every 24 hours. At these times, the DNS caches do not cache the requested domain names or has just reached a remaining TTL of 0. Between these peaks and from time to time, the adversary is able to set one or two cache entries. This is possible because the domain name selection process encountered some error such as a timeout while reversing an IP address or while requesting the reference TTL. The adversary will thus select a list of domain names different from the list selected by the requester. This has only a limited impact on the verification. A verifier relies on the domain names and resolver included in T and will not use the cache entries set by the adversary. Even if the verifier regenerates the list of domain names, it is unlikely that she will select a majority of cache entries the adversary has set. We verified the latter assertion by continuously executing the domain name generation process during the validity period (24 hours) of the initial timestamp and checking the remaining TTL. Figure 5.3b shows that the remaining TTL of 98% of the cache entries decrement exactly as expected and are not perturbed by the attack.

5.1.4 Conclusion

In this work we proposed a new trusted timestamping scheme, called DNStamp, that exclusively relies on the caches of the Domain Name System. DNStamp does not require a dedicated trusted service nor any form of collaboration among participants using the timestamping service. DNStamp can be used without registration to any dedicated service. Thus, anyone with Internet access can request and verify timestamps. Our experiments showed that we can set and reliably verify timestamps and that adversaries with reasonable capabilities cannot overwrite an existing timestamp.

Since our work, the concept of blockchain as used by the BitCoin protocol attracted a lot of attention [TS]. A blockchain can be used as a trusted timestamping service [CE12] and has the advantage that the associated timestamps are not short-lived. However, the size of a blockchain continuously increases, and this fact has already been pointed out as a potential issue in terms of

scalability and centralization risk [EGSVR16, Eth, CDE+16]. Further, the BitCoin blockchain only updates every 10 minutes which limits the granularity of timestamps. DNStamp may be used as a complement to BitCoin blockchains to solve some of the limitations of blockchains. E.g. DNStamp and blockchains together can provide fine-grained timestamp, by validating DNStamp timestamps and committing then into the blockchain. Being ephemeral, DNStamp also offers an alternative solution to blockchains for short-lived timestamps that do not need to be remembered forever.

5.2 Caching Attacks in the Cloud

Cloud computing leverages shared hardware to reduce infrastructure costs. The hypervisor, at the virtualization layer, provides isolation between the virtual machines. However, the last years have shown a great number of information leakage attacks across virtual machines, namely covert and side channels [IAIES14a, IAIES14b, RTSS09, WXW12, XBJ+11, YF14, ZJRR12]. These attacks violate the isolation. In this Section, we focus on covert channels that are rendered possible by CPU caches.

There are several challenges for covert channels across virtual machines. First, functions of address translation, and functions that map an address to a cache set are not exposed to processes, and thus induce uncertainty over the location of a particular data in the cache. This *addressing uncertainty* prevents the sender and the receiver to agree on a particular location to work on. Second, core migration drastically reduces the bitrate of channels that are not cross-core [XBJ+11].

Covert channels that don't tackle the *addressing uncertainty* are limited to use private first level caches, thus to be on the same core for modern processors [WXW12]. This dramatically reduces the bitrate in virtualized environment or in the cloud, with modern processors that have several cores with a shared and physically indexed last level cache. Ristenpart et al. [RTSS09] target the private cache of a core and obtain a bitrate of 0.2bps, with the limitation that the sender and receiver must be on the same core. Xu et al. [XBJ+11] quantify the achievable bitrate of this covert channel: from 215bps in lab condition, they reach 3bps in the cloud. This drop is due to the scheduling of the virtual machines across cores. Yarom and Falkner [YF14] circumvent the issue of physical addressing by relying on deduplication offered by the hypervisor or the OS. With deduplication, common pages use the same caches lines. However, deduplication is in general deactivated.

In this Section, we build a novel cross-core covert channel that uses the last level cache of the CPU. Our technique is able to bypass *addressing uncertainty*. Further, we introduce a generic method for mapping physical addresses to last level cache slices, allowing us to improve the previously introduced covert channel by a factor of 300 using the reversed function.

5.2.1 CPU cache fundamentals

The processor stores recently-used data in a hierarchy of caches to reduce the memory access time by the processor. The first two levels L1 and L2 are usually small and private to each core. The L3 is also called Last Level Cache (LLC). It is shared among cores and can store several megabytes. The LLC is *inclusive*, which means it is a superset of the lower levels.

Caches are organized in 64-byte long blocks called *lines*. The caches are *n-way associative*, which means that a line is loaded in a specific set depending on its address, and occupies any of the n lines. When all lines are used in a set, the *replacement policy* decides the line to be evicted to make room for storing a new cache line. Efficient replacement policies favor lines that are the least likely to be reused. Such policies are usually variations of Least Recently Used (LRU).

The first level of cache is indexed by virtual addresses, and the two other levels are indexed by physical addresses. With caches that implement a *direct addressing* scheme, memory addresses can be decomposed in three parts: the tag, the set and the offset in the line. The lowest $\log_2(\text{line size})$ bits determine the offset in the line. The next $\log_2(\text{number of sets})$ bits determine the set. The remaining bits form the tag.

The LLC is divided into as many slices as cores, interconnected by a ring bus. The slices contain sets like the other levels. An undocumented hashing algorithm determines the slice associated to an address in order to distribute traffic evenly among the slices and reduce congestion. In contrast to direct addressing, it is a *complex addressing* scheme. Potentially all address bits are used to determine the slice, excluding the lowest $\log_2(\text{line size})$ bits that determine the offset in a line. Contrary to the slices, the sets are directly addressed. Figure 5.4 gives a schematic description of the addressing of slices and sets.

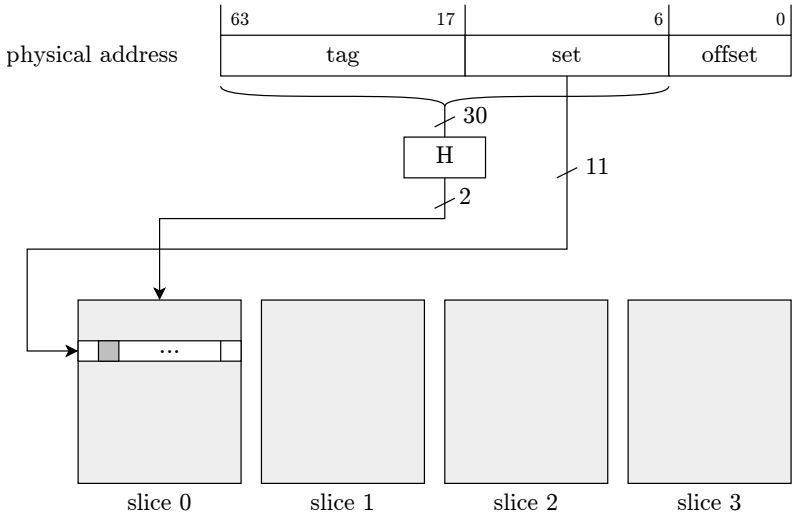


Figure 5.4: Complex addressing scheme in the LLC with 64B cache lines, 4 slices and 2048 sets per slice. The slice is given by a hash function that takes as an input all the bits of the set and the tag. The set is then directly addressed. The dark gray cell corresponds to one cache line.

5.2.2 C5 Covert Channel

Our covert channel relies on the fact that the LLC is shared and inclusive. Those two characteristics are present in all CPUs from Nehalem to Haswell microarchitectures, *i.e.*, all modern Intel CPUs, including most CPUs that are found in, e.g., Amazon EC2.

The sender process sends bits to the receiver by varying the access delays that the receiver observes when accessing a set in the cache. At a high level view, the covert channel encodes a '0' as a fast access for the receiver and a '1' as a slow access. In this sense, our covert channel strategy is close to *prime+probe*.

Figure 5.5 illustrates our covert channel. The receiver process repeatedly probes one set. If the sender is idle (a '0' is being transmitted), the access is fast because the data stays in the private L1 cache of the receiver, see Figure 5.5-1. The data is also present in the LLC because of its *inclusive* property.

To send a '1', the sender process writes data to occupy the whole LLC, see Figure 5.5-2; in particular this evicts the set of the receiver from the LLC. Because of the *inclusive* property, the data also gets evicted from the private L1 cache of the receiver. The receiver now observes that the access to its set is slow; the data must be retrieved from RAM, see Figure 5.5-3.

We now provide a detailed description of the sender and the receiver.

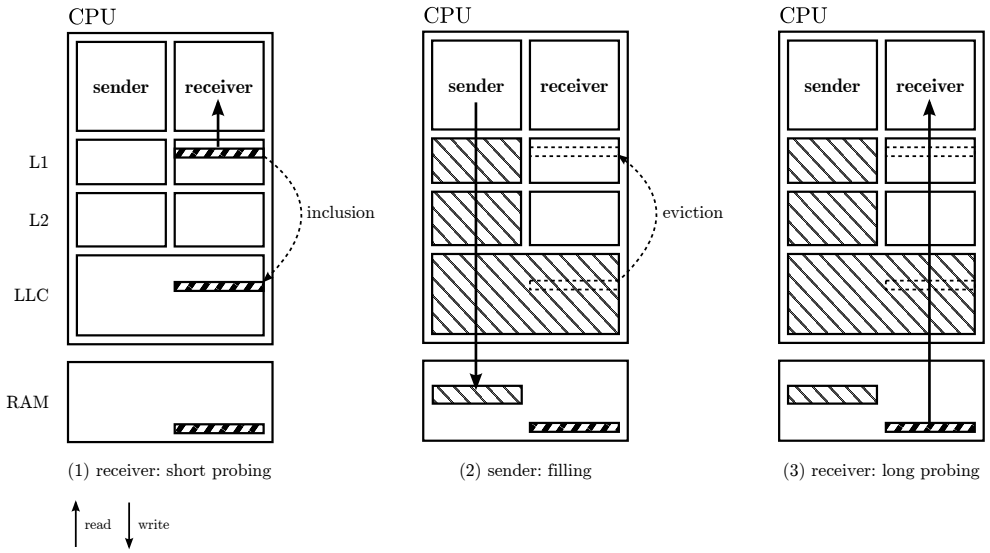


Figure 5.5: Cross-core covert channel illustration of sender and receiver behavior. Step (1): the receiver probes one set repeatedly; the access is fast because the data is in its L1 (and LLC by inclusive feature). Step (2): the sender fills the LLC, thus evicting the set of the receiver from LLC and its private L1 cache. Step (3): the receiver probes the same set; the access is slow because the data must be retrieved from RAM.

Sender

The sender needs a way to interfere with the private cache of the other cores. In our covert channel, the sender leverages the inclusive feature of the LLC (see Section 5.2.1). As the LLC is shared amongst the cores of the same processor, the sender may evict lines that are owned by other processes, and in particular processes running on other cores.

A straightforward idea is that the sender writes in a set, and the receiver probes the same set. However, due to virtualization and complex addressing, the sender and the receiver cannot agree on the cache set they are working on. Our technique consists of a scheme where the sender flushes the whole LLC, and the receiver probes a single set. That way, the sender is guaranteed to affect the set that the receiver reads, thus resolving the *addressing uncertainty*.

We leverage the replacement policy within a set to evict lines from the LLC. The replacement policy and the associativity influence the buffer size b of the sender. Considering a pure LRU policy, writing n lines in each set is enough to flush all the lines of the LLC, n being the associativity. The replacement

Algorithm 1 Receiver: $f(n, o, s)$

```
n ← L1 associativity
o ← log2(line size)
s ← log2(number of sets in L1)
buffer[n × 2^(o+s)]
loop
  read ← 0
  begin measurement
  for i = 0 to n do
    read += buffer[2^(o+s) i]
  end for
  end measurement, record (localTime, accessDelay)
end loop
```

policies on modern CPUs drastically affect the performance of caches; therefore they are well guarded secrets. Pseudo-LRU policies are known to be inefficient for memory intensive workloads of working sets greater than the cache size. Adaptive policies [QJP+07] are more likely to be used in actual processors. Since the actual replacement policy is unknown, we determine experimentally the size b of the buffer to which the sender needs to write.

The parameters used by the sender are the LLC associativity n, the number of sets 2^s, the line size 2^o, and a constant c to adapt the buffer size. To send a '1', the sender flushes the entire LLC by writing in each line j ($n \times c$ times) of each set i, using a specific pattern. Ideally, to iterate over the buffer we would take into account the function that maps an address to a set. However this function is undocumented, thus we assume a direct addressing; other types of iterations are possible. The sender writes with the following memory pattern $2^o i + 2^{o+s} j$. To send a '0', the sender does nothing. The sender waits for a determined time w before sending a bit to allow the receiver to distinguish between two consecutive bits.

Receiver

The receiver repeatedly probes all the lines of the same cache set in its L1 cache. Algorithm 1 summarizes the steps performed by the receiver. The iteration is dependent on the cache microarchitecture. To access each line i (n times) of the same set, the receiver reads a buffer – and measures the time taken – with the following memory pattern: $2^{o+s} i$. The cumulative variable `read` prevents optimizations from the compiler, by introducing a

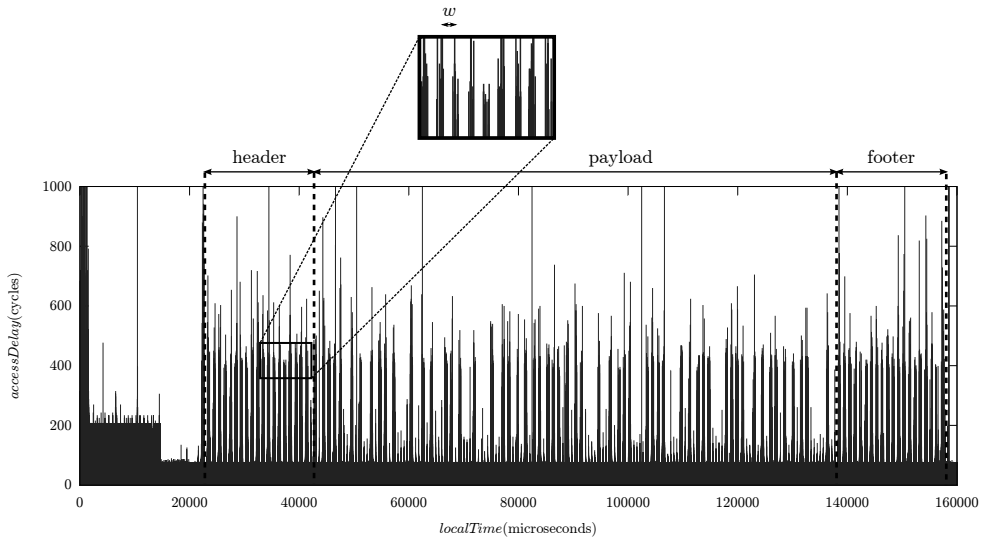


Figure 5.6: Reception of a 128-bit transmission. *Laptop* setup in native environment, with $w = 500\mu s$, $b = 3$MB.

dependency between the consecutive loads so that they happen in sequence and not in parallel. In the actual code, we also unroll the inner `for` loop to reduce unnecessary branches and memory accesses.

The receiver is able to probe a set in its L1 cache because the L1 is virtually indexed, and does not use complex addressing. We do not seek to probe the L2 or L3, because all read and write accesses reach the L1 first and they might evict each other, creating differences in timing that are not caused by the sender.

The receiver probes a single set when the sender writes to the entire cache, thus one iteration of the receiver is faster than one iteration of the sender. The receiver runs continuously and concurrently with the sender, while the sender only sends one bit every w microseconds. As a consequence, the receiver performs several measurements for each bit transmitted by the sender.

One measurement of the receiver has the form $(localTime, accessDelay)$, where $localTime$ is the time of the end of one measurement according to the local clock of the receiver and $accessDelay$ is the time taken for the receiver to read the set. Figure 5.6 illustrates the measurements performed by the receiver.

Having these measurements, the receiver decodes the transmitted bit-sequence. First, the receiver extracts all the '1's. The receiver removes all points that have an $accessDelay$ below (or equal to) typical L2 access time. Then the receiver only keeps the $localTime$ information and applies a clustering

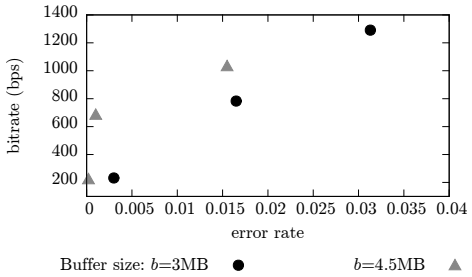


Figure 5.7: Bitrate as a function of the error rate, for two sizes of buffer ***b***. *Laptop* setup (3MB LLC) in native environment (no virtualization).

algorithm to separate the bits. We choose DBSCAN [EKSX96], a density-based clustering algorithm. Once all the '1's of the transmitted bit-sequence have been extracted, the receiver reconstructs the remaining '0's. This step is straightforward as the receiver knows the time taken to transmit a '0' which is w.

5.2.3 Experiments

We evaluate the C5 covert channel on native and virtualized setups. We adjust two parameters: the size b of the buffer that evicts the LLC, and the delay w between the transmission of two consecutive bits. The size b and the delay w impact the bitrate and the error rate of the clustering algorithm, as depicted in Figure 5.7.The precision of the clustering algorithm increases with the size b, however the bitrate is proportionally reduced. The size b is controlled by the multiplicative parameter c and must be at least the size of the LLC. The bitrate increases with lower values of w, but the precision of the clustering algorithm decreases.

To evaluate the covert channel, the sender transmits a random 4096-bit message to the receiver. We transmit series of 20 consecutive '1's as a header and a footer framing the payload to be able to extract it automatically. The receiver then reconstructs the message from its measurements. We run 10 experiments for each set of parameters, and calculate the bitrate and the error rate. We derive the error rate from the Levenshtein distance between the sent payload and the received payload. The Levenshtein distance is the minimum number of characters edits and accounts for insertions, deletions and bit flips.

We evaluate C5 in the *laptop* setup[1], in a native (non-virtualized) environment. We run the sender and the receiver as unprivileged processes. To demonstrate the cross-core property of our covert channel, we pin the sender and the receiver to different cores[2]. Figure 5.6 illustrates a transmission of 128 bits in the *laptop* setup, for $w = 500\mu s$ and $b = 3$MB.

Figure 5.7 presents the results in the *laptop* setup, for two values of b, and three values for waiting time w. For $b = 3$MB (the size of the LLC), varying w we obtain a bitrate between 232bps and 1291bps. The error rate is comprised between 0.3% (with a standard deviation $\sigma = 3.0 \times 10^{-3}$) and 3.1% ($\sigma = 0.013$). When we increase b to 4.5MB, the bitrate slightly decreases but stays in the same order of magnitude, between 223bps and 1033bps. The error rate decreases between 0.02% ($\sigma = 8.5 \times 10^{-5}$) and 1.6% ($\sigma = 1.1 \times 10^{-4}$). The standard deviation of the error rate also decreases, leading to more reliable transmission. We conclude that it is sufficient to write n lines per set, but that the transmission is more reliable if we write more than n lines. This is a tradeoff between the bitrate and the error rate.

We also evaluated C5 in a virtualized environment. The details can be read in the paper **[MNHF15]**.

5.2.4 Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters

A natural extension of our work on covert channels was to reverse engineer the complex cache addressing scheme of the cache. In **[MLN+15]** we develop a fully automatic approach to resolve the complex addressing of last level cache slices. Our technique relies on hardware performance counters - special-purpose registers that are used to monitor hardware-related events - to measure the number of accesses to a slice and to determine on which slice a memory access is cached. As a result, we obtain a translation table that allows determining the slice used by a given physical address (see example in Table 5.1). In the general case, finding a compact function from the mapping is NP-hard. Nevertheless in **[MLN+15]**, we describe an efficient algorithm to find a compact solution for a majority of processors (which have 2^n cores). We evaluate our method on processors of different micro-architectures with

[1] An i5-3340M Ivy Bridge processor with 2 cores and a 3MB LLC composed of 4096 sets. The LLC is 12-associative and uses complex addressing. The line size in all cache hierarchy is 64 bytes, and the L1 is 8-associative and has 64 sets.

[2] Using the `sched_setaffinity(2)` Linux system call.

Table 5.1: Mapping table. Each address has been polled 10000 times.

Physical address	CBo 0	CBo 1	CBo 2	CBo 3	Slice
0x3a0071010	**11620**	1468	1458	143	0
0x3a0071050	626	**10702**	696	678	1
0x3a0071090	498	567	**10559**	571	2
0x3a00710d0	517	565	573	**10590**	3
...	...	...	...	...	...

various numbers of cores. We do not detail the reverse engineering method in this document, but rather show how to use its results to speed-up the covert channel.

Instead of evicting the entire cache, having the complex addressing function, we can now target a particular set in a slice, and thus evict a cache line with much fewer accesses. Knowing the complex addressing function, in the case of a 12-way associative LLC, assuming a pseudo-LRU replacement policy, the sender only needs approximately 12 accesses to evict the whole set.

We conduct an experiment on the *laptop* setup in native environment and estimate the bitrate of this covert channel. The sender transmits interleaved '0's and '1's. According to the measurements, 29 bits can be transmitted over a period of 130 microseconds, leading to a bitrate of approximately 223 kilobits per second. This is a speedup of around 200 times compared to C5.

5.2.5 Conclusion

With the adoption of cloud computing and most services offered to end-used already run on shared and virtualized machines. While this has clear cost and performance benefits, it also exposes the application to potential information leakage attacks. In this Section, we built C5, a CPU cache-based covert channel that transfers messages across different cores of the same processor. Our covert channel tackles *addressing uncertainty* that is in particular introduced by hypervisors and complex addressing. In contrast to previous work, our covert channel does not require any shared memory. All these properties make our covert channel fast and practical.

We analyzed the root causes that enable this covert channel, *i.e.*, microarchitectural features such as the shared last level cache, and the inclusive feature of the cache hierarchy. We experimentally evaluated the covert channel in native and virtualized environments. We successfully established a covert channel between virtual machines despite the CPU scheduler of the hypervisor. We measured a bitrate one order of magnitude above previous cache based covert

channels in the same setup. Finally, we reverse engineered the complex addressing function of CPU caches and could further speedup the covert channel by a factor of 200.

Future research should investigate the possible countermeasures. These countermeasures may consist in partitioning the cache, *i.e.*, limiting the access to some portions of the cache for a particular application. It should also be investigated if an ongoing attack can be detected, e.g., by observing the number of cache accesses using the performance counters.

The covert channel presented in this chapter is just one instance of an attack rendered possible by shared caches. Many of such shared caches exists, especially in the cloud. Future work should investigate if other caches, e.g., disk caches, exhibit similar side channels. Other types of attacks should also be studied, e.g., attacks that impact the privacy of users of a cloud service.

6

Conclusion and future directions

This document presented several contribution to content placement, load-balancing and caching policies. We considered a large variety of applications, ranging from trusted timestamping to Video-On-Demand services. We also considered different infrastructures, ranging from edge networks devices (home gateways and set-top boxes) to the cloud. We also showed the potential security impacts of caching in the specific case of shared CPU caches.

Caches and load-balancing mechanisms are present in most distributed systems and this will not change in the future. Therefore this research topic will still be relevant several years from now. Future research will need to take into account the specificities of new architectures and services that will be offered to end-users. I now detail more specific research directions that I consider worthwhile investigating.

Software defined caches

Software defined networks (SDNs) [NMN+14] - one of the current trending topics - allow to programmatically control and manage networks, *i.e.*, define the network behavior, topology and routing through code. Caches are currently not considered in SDN architectures and are therefore not programmable. Still, a cache is an intrinsic part of the network, and should - in the SDN model - be programmable. More generally, the concept of programmable caches can also be transposed to other settings and caches, e.g., processes being able to program the cache of the CPU or of the hard disk. Such software defined caches have several advantages and open many new research directions. An

application, or an operator may programmatically define policies that are adapted to the network or to applicative specificities and tune the cache to a specific workload or with respect to a specific objective. While some application may program the cache to optimize the delay or the throughput, other applications can program the cache with respect to some security objective and some attacker model. E.g. sensitive application may be able to protect its cache (or specific items in the cache) from caching attacks. Policies can also be dynamically re-progammed according to changing conditions, e.g., a change in the workload or a change in the attacker model. Of course, programmable caches can also be used to perform attacks: by controlling the caching policy an attacker could implement fine-grained measurements for covert or side channels.

Machine learning of policies

Machine learning showcased significant advances in the last years and have proved to be efficient for a large set of problems. Despite the big hype around machine learning, little work [ASI12b, ASI12a, RE11, CE08] considered the usage of machine learning to automatically learn some caching, load balancing or data placement policy. Considering the advances in machine learning, I suspect that very efficient and adaptive policies can be designed. The machine learning models could be used to predict a particular workload, or as in recommender systems recommend the best caching policy given an observed workload. The models could further be continuously updated and reinforced (using reinforcement learning) according to the evolving system load and the caching performance. Software defined caches as described previously might be the enabler for machine learnt policies, as such caches allow to observe and re-programm caches dynamically.

Security and Privacy

Our particular instance of a CPU-cache covert channel in Chapter 5 demonstrates the potential security impact of shared caches. Side and covert channels exists in many other type of caches and are not restricted to CPU caches. E.g. DNS, HTTP and cloud-based caching as in Chapter 4 may leak information about their users. Further, cloud providers such as Amazon AWS add many new services to their offerings (e.g., Amazon Machine Learning, Amazon Elasticsearch etc.). These services rely on shared infrastructure; for each of these new services we can ask ourselves if there exists some sort of shared cache and if it can be exploited by an attacker. Future research should explore

these attacks. Further, while the cost and performance benefits of caching are often clear from the beginning, the security implications are more complicated to understand and model. Future research may help define proper models and methods in this direction. Software defined caches as described previously might also help in designing and programming security resistant caches.

Bibliography

[ACMR95] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 238–247, 1995.

[AGR06] S. Annapureddy, C. Gkantsidis, and P. Rodriguez. Providing Video-on-Demand using Peer-to-Peer networks. In *Internet Protocol TeleVision (IPTV) workshop*, 2006.

[ASI12a] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. Intelligent naïve bayes-based approaches for web proxy caching. *Knowledge-Based Systems*, 31:162–175, 2012.

[ASI12b] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. Intelligent web proxy caching approaches based on machine learning techniques. *Decision Support Systems*, 53(3):565–579, 2012.

[Ban12] Thepparit Banditwattanawong. From web cache to cloud cache. In *GPC'12*, 2012.

[BFD97] W. Bolosky, R. Fitzgerald, and J. Douceur. Distributed schedule management in the Tiger video fileserver. In *Proc. ACM SOSP*, 1997.

[bib] Cisco VNI Forecast and Methodology, 2015-2020. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html`. Updated June 01, 2016; Document ID 1465272001663118.

[BL07] James Bennett and Stan Lanning. The netflix prize. In *KDD cup and workshop*, 2007.

[BLM02] John W. Byers, Michael Luby, and Michael Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *IEEE J-SAC, Special Issue on Network Support for Multicast Communication*, 20(8), 2002.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Workshop on Bitcoin and Blockchain Research*, 2016.

[CDFK11] Claude Castelluccia, Emiliano De Cristofaro, Aurélien Francillon, and Mohamed-Ali Kaafar. EphPub: Toward robust Ephemeral Publishing. In *ICNP 11*, 2011.

[CE08] Jake Cobb and Hala ElAarag. Web proxy cache replacement scheme based on back-propagation neural network. *Journal of Systems and Software*, 81(9):1539 – 1558, 2008.

[CE12] Jeremy Clark and Aleksander Essex. CommitCoin: Carbon Dating Commitments with Bitcoin. In *FC'12*, 2012.

[CI97] P Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USITS'97*, 1997.

[CKR+07] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-yeol Ahn, and Sue Moon. I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System. In *IMC'07*, 2007.

[CRSZ01] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM SIGCOMM*, 2001.

[DLHC05] C. Dana, D. Li, D. Harrison, and C. Chuah. BASS: BitTorrent assisted streaming system for video-on-demand. In *IEEE MMSP*, 2005.

[EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI 16*, 2016.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Conference on Knowledge Discovery and Data Mining (KDD'96)*, 1996.

[ELZ86] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. software engineering*, SE-12(5):662–675, 1986.

[Eth] Etherum. White Paper - A Next-Generation Smart Contract and Decentralized Application Platform. `https://github.com/ethereum/wiki/wiki/White-Paper#scalability`. Retrieved August, 2016.

[FRRS12] Christine Fricker, Philippe Robert, James Roberts, and Nada Sbihi. Impact of traffic mix on caching performance in a content-centric network. In *INFOCOM Workshops'12*, 2012.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P'11*, 2011.

[HMNO12] Olivier Heen, Erwan Le Merrer, Christoph Neumann, and Stéphane Onno. Distributed and Private Group Management. In *IEEE International Symposium on Reliable Distributed Systems (SRDS 2012)*, 2012.

[HNMD12] Olivier Heen, Christoph Neumann, Luis Montalvo, and Serge Defrance. Improving the Resistance to Side-channel Attacks on Cloud Storage Services. In *IFIP International Conference on New Technologies, Mobility and Security (NTMS'12)*, 2012.

[HNO11] Olivier Heen, Christoph Neumann, and Stéphane Onno. État de l'art de la prise d'empreinte 802.11. In *Computer &Electronics Security Applications Rendez-vous (C&ESAR)*, 2011.

[HNO12] Olivier Heen, Christoph Neumann, and Stéphane Onno. Prise d'empreinte 802.11. *Misc Magazine*, 59, 2012.

[Hu92] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–61, 1992.

[IAIES14a] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain Cross-VM Attacks on Xen and VMware are possible! *Cryptology ePrint Archive, Report 2014/248*, 2014.

[IAIES14b] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14*, 2014.

[JGSW] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge. Hot Chips 2012, `http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf`. Retrieved July 2016.

[JSBM02] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Transactions on Networking*, 10(5), October 2002.

[JST+09] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *CoNext*, 2009.

[Kor97] J. Korst. Random duplicated assignment: An alternative striping in video servers. In *Proc. ACM Multimedia*, 1997.

[KPN+93] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. RFC1536 - Common DNS Implementation Errors and Suggested Fixes, 1993.

[LC97] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *ICDCS'97*, 1997.

[Lub02] Michael Luby. LT Codes. In *Proc. IEEE Symposium on foundation of computer science (FOCS)*, 2002.

[LUKM06] Arnaud Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *ACM SIGCOMM IMC*, 2006.

[**MLN+15**] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, 2015.

[MM03a] P. Maymounkov and D. Mazieres. Rateless codes and big downloads. In *Proc. the International Workshop on Peer-to-Peer Systems*, February 2003.

[MM03b] N Megiddo and DS Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST'03*, 2003.

[MNHF14] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality Issues on a GPU in a Virtualized Environment. In *Financial Cryptography and Data Security (FC'14)*, 2014.

[**MNHF15**] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5 : Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015. Best Paper Award.

[MON+13] Clémentine Maurice, Stéphane Onno, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Improving 802.11 Fingerprinting of Similar Devices by Cooperative Fingerprinting. In *Conference on Security and Cryptography (SECRYPT 2013)*, 2013.

[MRS01] M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices: A survey of the techniques and results. *Handbook of Randomized Computing*, 1:255–305, July 2001.

[MTS90] R. Mirchandaney, D. Towsley, and J. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of parallel and distributed computing*, 9:331–346, 1990.

[NHO12] Christoph Neumann, Olivier Heen, and Stéphane Onno. An empirical study of passive 802.11 Device Fingerprinting. In *IEEE ICDCS Workshop on Network Forensics, Security and Privacy (NFSP'12)*, 2012.

[**NHO14**] Christoph Neumann, Olivier Heen, and Stéphane Onno. DNStamp : Short-lived Trusted Timestamping. *Computer Networks*, 64, 2014.

[NMN+14] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[NPVS07] Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *ACM SIGCOMM Computer Communication Review*, 37(1), 2007. Invited Paper.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In *Selected areas in cryptography (SAC'06)*, 2006.

[OGHN12] Stéphane Onno, Raphael Gelloz, Olivier Heen, and Christoph Neumann. User-Based Authentication for Wireless Home Networks. In *IEEE International Conference on Consumer Electronics Berlin (ICCE Berlin)*, 2012.

[ONH12] Stéphane Onno, Christoph Neumann, and Olivier Heen. Conciliating remote home network access and MAC-address control. In *IEEE International Conference on Consumer Electronics (ICCE 2012)*, 2012.

[OOW93] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD'93*, 1993.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA 2006*, 2006.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[PLS+14] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *ASIACCS 2014*, 2014.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381, 2007.

[RE11] Sam Romano and Hala ElAarag. A neural network proxy cache replacement strategy and its implementation in the squid proxy server. *Neural Computing and Applications*, 20(1):59–78, 2011.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS'09*, 2009.

[RV00] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8, 2000.

[SBI05] A. Sharma, A. Bestavros, and I.Matta. dPAM: a distributed prefetching protocol for scalable asynchronous multicast in P2P systems. In *INFOCOM*, 2005.

[SDK+06] K. Suh, C. Diot, J.F. Kurose, L. Massoulié, C. Neumann, D. Towsley, and M. Varvello. Push-to-peer video-on-demand system: design and evaluation. Technical Report CR-PRL-2006-11-0001, Thomson Research, nov 2006.

[**SDK+07**] Kyoungwon Suh, C. Diot, J. Kurose, L. Massoulie, Christoph Neumann, D. Towsley, and Matteo Varvello. Push-to-Peer Video-on-Demand System: Design and Evaluation. *IEEE Journal on Selected Areas in Communications*, 25(9), 2007.

[**SNS14**] Nicolas Le Scouarnec, Christoph Neumann, and Gilles Straub. Cache policies for cloud-based systems : To keep or not to keep? In *IEEE International Conference on Cloud Computing (CLOUD 14)*, 2014. Best Paper Award.

[SRT99] S.Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *INFOCOM*, 1999.

[TK07] Saurabh Tewari and Leonard Kleinrock. Analytical model for BitTorrent-based live video streaming. In *IEEE NIME Workshop*, 2007.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, July 2010.

[TS] Florian Tschorsch and Björn Scheuermann. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies.

[VIF06] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for supporting streaming applications. In *INFOCOM*, 2006.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[XBJ+11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Cloud Computing Security Workshop (CCSW'11)*, 2011.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[YYLC10] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *IPDPS'10*, 2010.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS'12*, 2012.

[ZLLY05] X. Zhang, J. Liu, B. Li, and T. Yum. CoolStreaming/DONet: A data-driven overlay network for live media streaming. In *INFOCOM*, 2005.