Pollution Attacks and Defenses for Internet Caching Systems ⋆

Leiwen Deng, Yan Gao, Yan Chen and Aleksandar Kuzmanovic

Northwestern University, Department of EECS
2145 Sheridan Road, Evanston, IL, 60208, USA.

Abstract

Proxy caching servers are widely deployed in today's Internet. While cooperation among proxy caches can significantly improve a network's resilience to denial-of-service (DoS) attacks, lack of cooperation can transform such servers into viable DoS targets. In this paper, we investigate a class of pollution attacks that aim to degrade a proxy's caching capabilities, either by ruining the cache file locality, or by inducing false file locality. Using simulations, we propose and evaluate the effects of pollution attacks both in web and peer-to-peer (p2p) scenarios, and reveal dramatic variability in resilience to pollution among several cache replacement policies.

We develop efficient methods to detect both false-locality and locality-disruption attacks, as well as a combination of the two. To achieve high scalability for a large number of clients/requests without sacrificing the detection accuracy, we leverage streaming computation techniques, *i.e.*, bloom filters and probabilistic counting. Evaluation results from large-scale simulations show that these mechanisms are effective and efficient in detecting and mitigating such attacks. Furthermore, a Squid-based implementation demonstrates that our protection mechanism forces the attacker to launch extremely large distributed attacks in order to succeed.

Key words: proxy-cache-targeted, locality-disruption, false-locality

1 Introduction

Caching has proven to be one of the most valuable and widely-applied techniques in computer systems. The idea is simple: the first time a query is run

⋆ A subset of this work appears in the Proceedings of *IEEE ICNP 2006* [1].

Email addresses: karldeng@cs.northwestern.edu (Leiwen Deng), ygao@cs.northwestern.edu (Yan Gao), ychen@cs.northwestern.edu (Yan Chen), akuzma@cs.northwestern.edu (Aleksandar Kuzmanovic).

for data, the results are saved in a cache; the next time the query is run for the same data, the data are retrieved from the cache without going to higher-latency memory. Because it can significantly enhance overall system performance, the same idea has been widely applied in the Internet. Instead of retrieving data from a distant server, the data can be retrieved from a proxy cache. This decreases the number of requests arriving at servers, reduces the amount of traffic in the network, and improves the client-perceived latency.

Unfortunately, like other systems in today's Internet, proxy caches can be used as efficient *tools* in the hands of malicious users. In particular, open proxy caches can invite traffic from malicious clients for various abuse-related activities: spamming, bulk data transfers, unauthorized downloading of licensed content, or for originating malicious outbound requests from the proxy [2]. However, little attention is given to scenarios in which proxy caches, both opened and closed, themselves can become *victims* of malicious clients.

In contrast to the thriving Content Distribution Network (CDN) business (*e.g.*, Akamai [3]), which is based on *server-side* cooperative caching, such cooperation is largely nonexistent at the client side. Thus, such proxies are highly vulnerable to DoS attacks in which the caching mechanism itself can become the primary target of the attack. This holds not only for widely-deployed web proxy caches, but also for thriving peer-to-peer (p2p) proxy caches. [4] shows that with a relatively small cache size, less than 10% of the total traffic, byte hit rate of P2P system is up to 35%. Recently, ISPs started caching p2p content at their boundaries [5], as p2p traffic accounts for the vast majority of the network's total bandwidth [6].

In this paper, we propose and study a class of *pollution* attacks targeted against Internet proxy caches. The attackers' goal is to severely degrade the caching service by polluting the cache with unpopular content. Even a single cache-miss to a large object can often require a large amount of data to be fetched from its distant origin server. A larger number of cache misses can further congest the network access link, particularly when p2p caches are targeted. Even a moderate degradation of the hit ratio can cause additional *hundreds of TBytes* of data to be transferred over an access link on a daily basis. Long periods of severe service reduction, in which both traffic load and file download time increase by several *orders of magnitude*, can on average degrade service more than classical high-rate DoS attacks are capable of. Moreover, the proposed attacks can be launched against low-level DNS servers. In such a scenario, a set of malicious attackers may pollute the local DNS server's cache with unpopular entries, thus significantly reducing the performance experienced by regular clients.

The proposed pollution attacks pose a challenging problem for the entire Internet community. First, such attacks have stealthy nature: they are capable

of degrading overall network performance without flooding network resources. Second, they possess a dangerous level of indirection: while both clients and servers are affected by the attack — neither clients nor servers are directly attacked. Third, they pollute the cache with *unpopular*, rather than bogus files, making them much harder to detect. Finally, no counter-pollution mechanisms exist in Internet caches; thus, even simple, brute-force pollution attacks can be quite successful. Indeed, while some Internet caching systems do apply simple mechanisms to mitigate the effects of *unintentional* cache pollution, we demonstrate that such mechanisms are fundamentally limited in their ability to thwart systematic, *intentional* pollution attacks; while being much more effective, such attacks are much harder to detect.

We propose and analyze two generic classes of attacks: locality-disruption and false-locality attacks. *Locality-disruption* attacks continuously generate requests for new unpopular files, thus ruining the cache file locality. *False-locality* attacks repeatedly request the same set of files, thus creating a false file locality at proxy caches. We conduct an extensive set of simulations, both with p2p and web workloads. To accurately represent the effects of pollution attacks, we define a metric, *byte damage ratio*, which successfully summarizes multiple statistics in the presence and absence of attacks.

Further, we demonstrate that the cache resilience to pollution attacks fundamentally depends on the replacement algorithm deployed. However, we show that a replacement algorithm alone is not capable of fully protecting the system against pollution attacks. In particular, we isolate scenarios in which all examined replacement algorithms exhibit extreme vulnerability to a subclass of pollution attacks. The Greedy Dual-Size Frequency (GDSF) algorithm is vulnerable to a class of low-rate size-targeted locality-disruption attacks. For example, an aggregate attacker rate that represents approximately 4% of the total network bandwidth is capable of almost fully degrading the caching service. The Least Frequently Used (LFU) algorithm is vulnerable to a class of false-locality attacks. And the Least Recently Used (LRU) replacement scheme exhibits a vulnerability to a class of false-locality attacks that exploit the strong seasonal behavior of the examined Internet content delivery systems.

The cache pollution attacks are very stealthy because they can be easily mixed with and regarded as normal clients' requests. Thus, no existing schemes is capable of detecting such attacks. We propose two cache pollution detection mechanisms to detect false-locality and locality-disruption attacks, respectively. Both are based on the inherent features of each attack. For the former, the key metric is the repeated requests from a relatively small set of clients (attackers) which are essential to keep the false locality of unpopular files. For the latter, the key metric is the small average duration/life-time of files in cache and a relatively small number of clients (attackers) who make these requests.

These two schemes can be further combined to detect an even more stealthy combination of false-locality and locality-disruption attacks. Since large ISPs may have millions of clients and/or millions of cached files, we further leverage data streaming computation techniques, *bloom filters* [7] and *probabilistic counting with stochastic averaging (PCSA)* [8,9] to significantly reduce the amount of state needed to maintain for detection. It not only improves the detection scalability, but also makes the detection system itself resilient to DoS attacks. Once any attack is detected, we ignore the requests from attackers and/or remove cached files with "false locality" for mitigation.

Simulation with large traces (100,000 clients and 100,000 files) show that we can effectively detect the intrusions unless there is a very large number of attackers or the attack damage is very limited. The bloom filters and PCSA dramatically reduce the memory consumption without sacrificing the accuracy.

2 Motivation and Scope

In this section, we present pollution scenarios, attack classes, and targeted replacement algorithms.

2.1 Pollution-Attack Scenarios

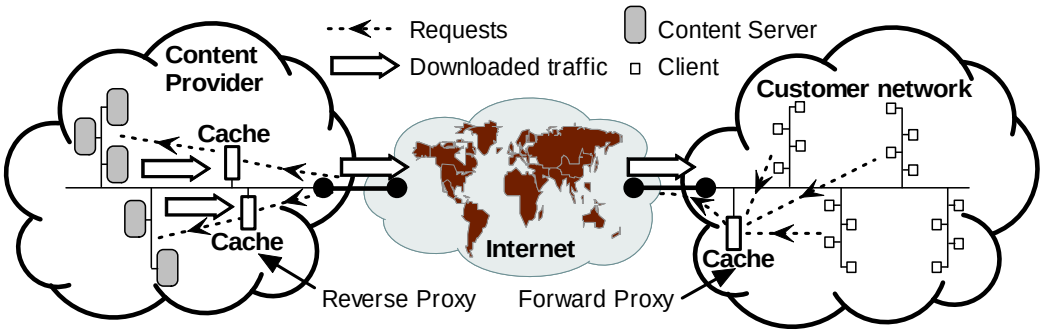
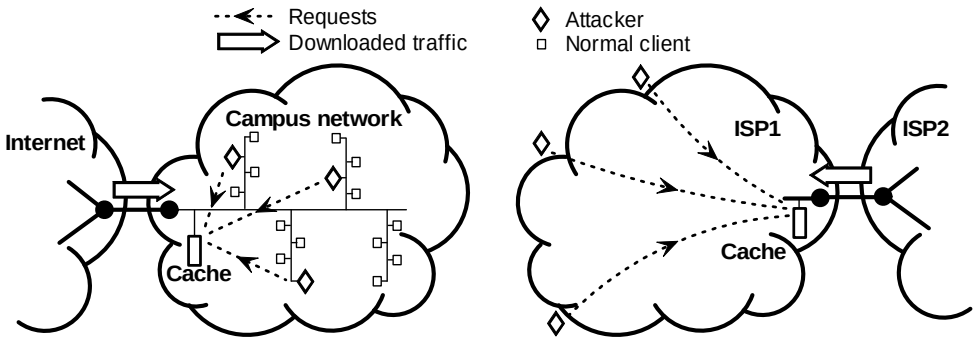


Fig. 1. Forward and reverse proxies

In general, there are two types of proxies: forward proxies and reverse proxies as shown in Figure 1. The former is the most common proxy: it performs as an intermediate server that sits between the client and the origin server. The latter appears to the client just like an ordinary web server. It is usually located close to a back-end server behind a firewall and provides Internet users access to such a server. Many popular websites nowadays rely heavily on reverse proxies when provisioning their back-end servers. Although polluting a reverse proxy cache tends to be technically more difficult than polluting a

forward proxy, polluting reverse proxies is highly feasible. We will focus on the forward proxies in this paper though the pollution techniques proposed apply well to both. In Section 4, we will discuss their differences and implications.



(a) Attacking a web cache (b) Attacking an ISP cache

Fig. 2. Proxy-cache-targeted pollution attacks

Here, we explore proxy-cache-targeted attack scenarios. The first is shown in Figure 2(a). An attacker can compromise a number of machines on an institutional or a corporate network and launch malicious requests to pollute the cache with unpopular content. The result of the attack is a lower hit ratio for legitimate clients, leading to reduced performance. Even a single cache-miss to a large file can cause large data transfers from distant origin servers. In such a scenario, the bottleneck capacity may be reduced by several orders of magnitude (*e.g.*, from 1 Gbps to less than 1 Mbps), thus dramatically degrading file-download performance.

A larger number of cache misses can further congest the network access link. More seriously, a larger-scale deployment of such attacks can cause an uncontrolled increase of the Internet traffic volumes and more frequent flash crowds at servers [10]. For example, a report on NLANR web caches pointed out that inappropriate settings of `Expires` and `Last-Modified` header fields prevent ordinary caches from effectively dealing with flash crowds [11].

While there is no evidence that proxy-cache-targeted pollution attacks are actively conducted in the Internet, another scenario is depicted in Figure 2(b). It is related to the dispute between the music industry and p2p file-sharing networks over the copyrighted content distributed on these networks. Recently, the music industry abandoned purely legal actions [12], and started launching denial of service attacks against p2p networks [13,14]. At the same time, ISPs started *caching* p2p content at their boundaries as p2p traffic accounts for the vast majority of the network's total bandwidth [6]. However, caching copies of pirated files made ISPs accomplices in illegal file trading, at least according to the music industry views [5]. Given the aggressive behavior that the music industry demonstrated against p2p file-sharing networks, it would be no surprise if the former started launching cache-targeted pollution attacks against

the ISP caches.

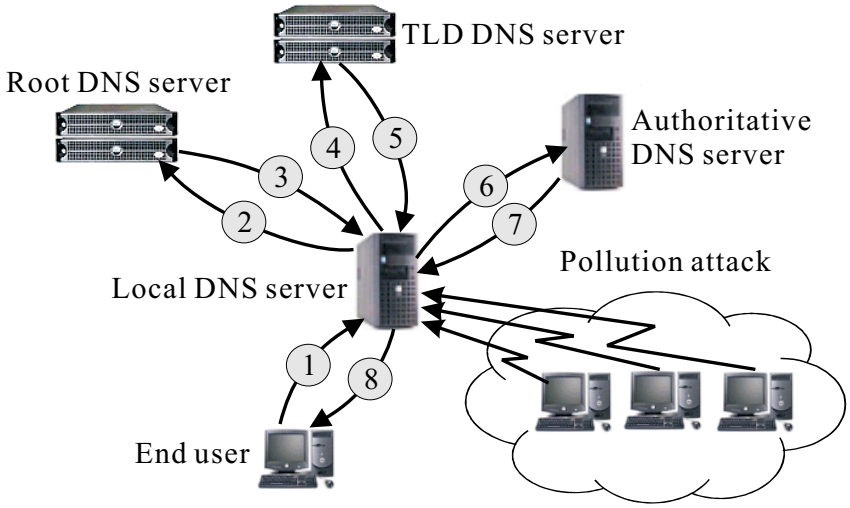


Fig. 3. Pollution attack against a local DNS server

Another scenario is the potential pollution attack against local (low-level) DNS servers as shown in figure 3. Such servers resign in each ISP and they are central to the DNS architecture. A set of malicious attackers may pollute the local DNS server's cache with unpopular entries, thus significantly reducing the performance experienced by regular clients. As shown in the figure, a cache miss will introduce a series of iterative queries to root, top-level-domain, and authoritative DNS servers. This adds annoying initial delays to web clients, significantly degrading their browsing experiences. Moreover, the study from [15] showed that hit ratios at low-level DNS servers are typically over 80%. Thus, reducing these ratios can dramatically overload low-level DNS servers, additionally impacting system's performance.

There are currently over 87 millions valid DNS entries [16], corresponding to *GBytes of data*, while typical cache size of local DNS servers is relatively small[17]. Even if we adopt the recent ideas to store *all* DNS entries in the cache [18], pollution attacks are still feasible. This is because the number of possible entries to be *cached* is infinite due to the following two reasons. One is the use of *wildcard* – a special record that is setup to resolve any query that does not match an existing authoritative record in the zone. The other is the use of *negative caching* – a server caches the resulting name error when a queried name does not exist. Thus, cache pollution attacks are possible no matter how large the system resources are. In this paper, we do *not* evaluate pollution attacks against DNS servers, simply because we were unable to find a representative workload for the DNS traffic. Still, we demonstrate below that our results are largely independent of the workload, thus generally applicable to all caching systems, including DNS.

Cooperation among proxies can improve a network's resilience to DoS at-

tacks [19]. However, in the absence of such cooperation, proxy caches themselves can become DoS targets. In contrast to the thriving *server-side* cooperative caching (*e.g.*, Akamai [3]), the client side largely lacks such cooperation. Despite the initial enthusiasm in developing hierarchical web caching systems (*e.g.*, [20–23]), the lack of trust among different institutions and companies has impeded deployment of client-side cooperative caching [24]. Another reason is the fact that the benefits of such cooperation are quite limited; in particular, it has been shown that the hit ratio increases only logarithmically (*i.e.*, very slowly) with the client population [15,25]. Finally, deploying a non-distributed proxy cache at an ISP edge is much simpler than deploying a proxy network. Thus, proxy caches are typically isolated, making them particularly vulnerable to DoS attacks.

2.2 Pollution-Attack Classes

By recognizing generic strategies that the attacker can exploit to pollute a cache with unpopular content, we characterize pollution attacks into the following categories: (*i*) locality-disruption and (*ii*) false-locality attacks.

Locality-disruption attacks aim to degrade cache efficiency by ruining its file locality. An attacker continuously generates requests for *new* unpopular files and disrupts the correlation structure of the original arrival request stream; this alters the cache contents, decreases the hit ratio experienced by regular clients, and eventually degrades their performance. For example, robots and crawlers deployed by search engines have a referencing pattern that can completely disrupt the locality assumptions and significantly increase the miss ratio of a *server-side* cache [26]. We will show below that *malicious* crawlers are capable of disrupting the *proxy-based* caches even more severely.

False-locality attacks aim to degrade the hit ratio experienced by regular clients by repeatedly requesting the same set of files, thus creating a false file locality at proxy caches. The key advantage of this attack is the ability to quickly refresh the polluted files in the cache. For example, consider the LAN scenario depicted in Figure 2(a). Assume that the LAN link rate is 1 Gbps, the access link rate is 100 Mbps, and the cache size is 100 GBytes. In such a scenario, it takes approximately 2-4 hours for malicious clients to populate the cache with irrelevant content, and only around *10 minutes* to fully refresh the entire polluted content in the cache. [1] Thus, once the attackers manage to "freeze" the cache, it becomes easier to keep it in such a state.

[1] When initially polluting the cache, the attacker is limited by the access link rate. However, when refreshing the content, it is limited by the local LAN rate, 1 Gbps in our scenario.

Both pollution-attack classes presented above require attackers not only to generate requests, but also to fully download the requested files. While it may appear attractive for the attackers to generate "drop connection" attacks,[2] such attacks are not feasible. This is because caching servers (*e.g.*, Squid [27]) buffer no more than 16 kB ahead of what has been seen by the client, and also *evict* files that have not been fully downloaded by clients.

2.3 Targeted Replacement Algorithms

A replacement algorithm defines which of the cached files is replaced by a new one, when a new file is added to a full cache. Below, we present cache-replacement algorithms widely deployed in the Internet today, whose resilience to pollution attacks we evaluate later in the paper.

The two most popular caching policies are *Least Recently Used* (LRU) and *Least Frequently Used* (LFU) [28]. LRU evicts the least recently accessed document first, on the basis the traffic exhibits temporal locality. Intuitively, the farther in time a document has last been requested, the less likely it will be requested in the near future. On the other hand, LFU evicts the least frequently accessed document first, on the basis that a popular document tends to have a long-term popularity profile.

In addition, we also evaluate the *Greedy Dual-Size Frequency* (GDSF) replacement policy [27,29,30]. GDSF discriminates against large documents, allowing for smaller documents to be cached. It also uses a dynamic aging policy at the same time.

Beyond the fact that all three replacement algorithms are operational in the current Internet [27,29,30], GDSF policy is of particular interest. This is because it employs a dynamic aging policy, which addresses the problem of *unintentional* pollution; such pollution arises when old popular objects reside for a long time in a cache and degrade the hit ratio. For example, pages that are accessed a large number of times during flash-crowd events may remain in the cache long after their popularity expires. Below, we explore the performance of the aging mechanism in presence of *intentional* pollution attacks. Finally, we examine the pollution-resilience properties of the three replacement policies both for web and p2p workloads.

[2] In such scenarios, an attacker would drop a connection soon after requesting a file, thus forcing only the cache server to download the file.

3 The Effects of Pollution

We present an extensive set of simulation experiments to explore the key system factors that influence pollution resilience of Internet caching servers.

3.1 Experimental Methodology

Simulator. We implement a discrete-event simulator for a caching system with the following capabilities/parameters: (1) support for LFU, LRU, and GDSF replacement algorithms, (2) variable cache size, (3) multiple DoS behaviors, and (4) multiple workloads characterizing behavior of regular clients. In addition, we simulate the effects of *access* and *local* network capacities on system performance. The access link capacity refers to the link capacity that is utilized in the case of cache misses, *e.g.*, a link between the two edge routers depicted in Figure 2(a). The local link capacity refers to the network capacity "in front of the cache", *e.g.*, the campus network in Figure 2(a). It is utilized both in cases of cache hits and misses. Discrepancy between the two capacities impacts the effectiveness of pollution attacks in a non-trivial way, as we explain in detail below.

Workloads. We generate p2p workloads by utilizing a model from [6]. Although this model is evaluated by only using file-sharing traffic, it is based on features (*e.g.*, fetch at most once) common for many other P2P traffic types (*e.g.*, video, bit torrent). As such, it characterizes P2P traffic in a more general way. The model captures key parameters of p2p workloads, such as request rates, number of clients, number of objects, and changes to the set of clients and objects. In addition, we generate web workloads by applying a version of the same model and by fitting a set of empirically-extracted distributions [31]. We also integrate parameters for network address translators (NAT) and time-of-day effects in our model. By varying model parameters, we are able to change workload characteristics (*e.g.*, hit ratio, aggregate bit rate) and to explore how changes to the parameters affect system's resilience to pollution attacks. Table 1 summarizes the typical settings of parameters.

Our web workload is generated as follows. Web clients may fetch a popular page (*e.g.*, Google) thousands of times; this behavior is best modeled by *fetch-repeatedly* systems [6]. We generate web object sizes by fitting the empirically-measured heavy-tailed distribution reported in [31]. While the majority of files are very short, such that the mean file size is approximately 7kBytes, web files on the order of GB's in size are also generated. Clients select objects from a Zipf distribution in an independent and identically distributed fashion. By changing the Zipf parameter α, we are capable of controlling the correlation

Parameters	p2p	web
Number of objects	10000	5000000
Average object size	50 MB	10 kB
Number of clients	2000	10000
Aggregate request rate (objects/hour)	180∼900	900k∼4.5M
New object arrival rate (objects/month)	10	n/a
Zipf parameter α	0.95	0.65
Percentage of clients behind NAT boxes		20%
Maximum multiplex ratio of every single NAT IP address		10

Table 1
Model parameters for regular clients

structure of the web request stream. Finally, we model the client request rates according to a heavy-tailed distribution, which is extracted from a representative web trace [32,33]. About 85% clients have a request rate lower than the *average* rate, yet 1% clients have a request rate 50∼85 times that of the *average*. This distribution corresponds to the regular-client behavior.

P2p and multimedia workloads in general are best modeled by *fetch-at-most-once* systems [6]. While the underlying popularity of objects in such systems is still driven by Zipf's law, the resulting workload does not follow Zipf any longer due to the *fetch-at-most-once* effect. [3] We distribute p2p file sizes according to the empirically-measured distribution reported in [6]; the majority of the files (approximately 90%) are smaller than 10 MBytes, yet very large files (*e.g.*, larger than 100 MBytes) are also fetched. We model the clients' request rate according to measurements reported in [34]. The corresponding distribution has the following characteristics: about 90% clients have a request rate lower than the *average*, yet 1% clients have a request rate exceeding $8 \times average$, 0.1% clients exceeding $36 \times average$ and 0.01% clients exceeding $400 \times average$. However, 50% clients have a request rate lower than $\frac{1}{40} \times average$. Finally, by varying the number of clients and objects in the system, we are capable of tuning the nominal network load and hit ratio experienced by clients, as we explain in detail below.

Methodology. Our main goal is to gain basic understanding about the effectiveness of the proposed cache-targeted pollution attacks. Thus, we focus on the *primary* consequences of the attack — the degradation of the hit ra-

[3] P2p clients rarely fetch the same object more than once. As a result, the creation of new objects, and the addition of new clients to the system are the primary forces that drive multimedia workloads; hence, the corresponing distribution does not follow Zipf's law.

tio during attacks. Moreover, we explore the *aggregate* malicious and regular clients' request rates as the key system parameters that influence the effects of pollution attacks.

To summarize multiple statistics in the presence and absence of attacks, we define the *byte damage ratio* as the key measure of the effectiveness of the attack. It is defined as $(BHR(n) - BHR(a))/BHR(n)$, in which $BHR(n)$ and $BHR(a)$ denote the byte hit ratios of regular clients in the absence/presence of an attack respectively. When the byte damage ratio is zero, the attack is completely ineffective; when the byte damage ratio equals one, the caching service is completely overrun. [4]

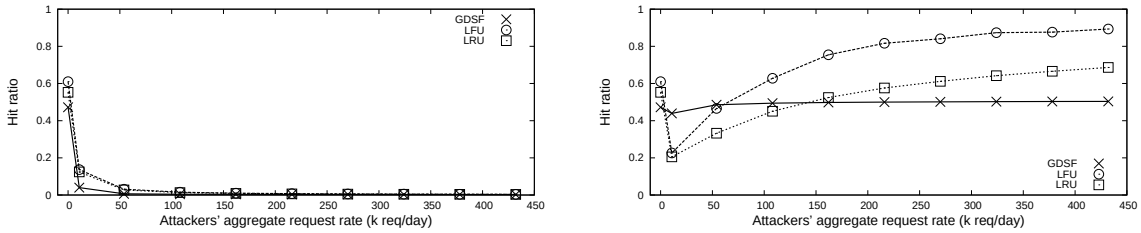
3.2 Baseline Experiments

We first consider a baseline scenario for two classes of pollution attacks described in Section 2.2: the locality-disruption and false-locality attacks. The majority of our experiments reveal similar trends both for p2p and web workloads. Thus, we present p2p results by default, and show web results only when they indicate different trends. Unless otherwise indicated, the average aggregate throughput generated by p2p clients is 20 Mbps, the cache size is set to 100 GBytes, and the Zipf parameter α is set to 0.95. In addition, attackers' *aggregate* request rate remains constant over time in all experiments; however, the *per-client* request rate is not uniform, and deviates based on the parameter setting. Table 2 summarizes parameter settings for the attacker.

Parameters	p2p	web
Number of bogus objects	1k (*FL*) 100k (*LD*)	10M (*FL*) 100M (*LD*)
Average object size	10 MB	10 kB
Number of clients	1∼100	1∼100
Aggregate request rate (objects/hour)	900∼45k	900k∼45M
$\frac{\textit{Deviation of request rate among clients}}{\textit{Average per-client request rate}}$		0.8

FL - false locality attack, LD - locality disruption attack

Table 2
Model parameters for attacker clients



(a) Locality-disruption attack (b) False-locality attack

Fig. 4. Total hit ratio in presence of attacks

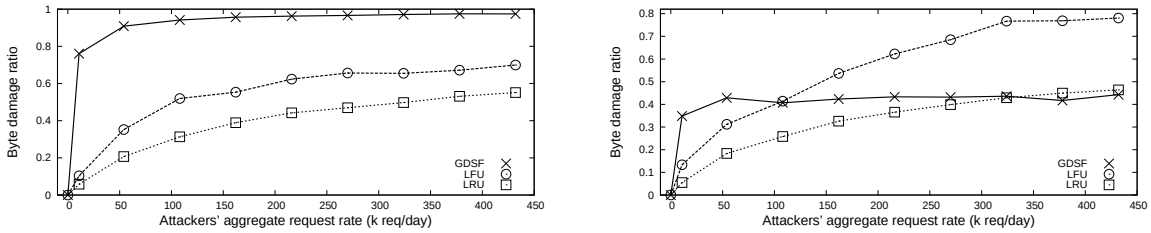
3.2.1 Locality-disruption and false-locality attacks

Locality-disruption attackers ruin the cache file-locality by generating malicious requests for new unpopular files. Figure 4(a) depicts the effects of the attack on the total hit ratio (of both malicious and regular clients) as a function of the aggregate malicious clients' request rate. The total hit ratio dramatically degrades in the presence of the attack. For example, when the malicious request rate changes from 0 to only 10 k req/day, the total hit ratio reduces significantly; when the malicious request rate is 50 k req/day, the total hit ratio becomes almost zero. As a point of reference, the aggregate request rate of regular clients is approximately 250 k req/day. A small percent of malicious requests (*e.g.*, below 4% of total number of requests) is capable of significantly degrading the overall hit ratio. This is because the attackers simultaneously (*i*) increase the total number of cache misses, and (*ii*) pollute the cache with unpopular content. While the degradation of the total hit ratio may appear to be a reliable attack signature, this is not the case with false-locality attacks.

False-locality attackers request unpopular files from a predetermined list. In this way, they create a false file-locality at the cache, and consequently degrade the hit ratio experienced by regular clients. Figure 4(b) depicts the effects of the attack on the *total* hit ratio as a function of the aggregate attackers' request rate. Contrary to the locality-disruption scenario, the total hit ratio *increases* when the attackers' aggregate request rate increases. Indeed, when the attackers' request rate is high, they manage to populate the cache with unpopular content. Consequently, the total number of cache hits increases because the number of "false" hits increases; at the same time, the number of "true" hits dramatically decreases, as we indeed demonstrate below. Thus, the *total* hit ratio cannot be used as a good indicator that the attack is taking place. For example, in the LFU case, the total hit ratio is 0.6 both in the absence of attack, and when the system is severely attacked (aggregate attackers' rate is 100 k req/day).

Figure 4(b) indicates initially, when the aggregate attackers' request rate is

[4] A negative byte damage ratio refers to the abnormal scenario in which the byte hit ratio of regular clients improves in presence of attack.



(a) Locality-disruption attack (b) False-locality attack

Fig. 5. Byte damage ratio

low (*e.g.*, 10 k req/day), the total hit ratio decreases. In such a scenario, the attackers do not manage to successfully build a false locality. This is because false unpopular files are evicted from the cache due to requests generated by regular files. Thus, re-requesting previously evicted files is similar to performing the locality-disruption attack. Consequently, the total hit ratio initially decreases as in the case of the locality-disruption attacks. Finally, Figure 4(b) reveals a dramatic difference in the performance of various cache replacement algorithms. We analyze this issue in detail below.

3.2.2 Replacement Algorithms

Here, we evaluate the impact that the cache replacement algorithms have on the resilience to pollution attacks. A replacement algorithm determines which of the cached files is evicted from the cache when a new file is added to a full cache. As a measure of the effectiveness of the attack, we apply the byte damage ratio of regular clients, defined above.

Figure 5(a) shows the byte damage ratio of regular clients as a function of the aggregate attackers' request rate. The most stunning result is the extreme vulnerability of the GDSF algorithm to low-rate pollution attacks. When the aggregate attackers' rate is as low as 10 k req/day, the byte damage ratio is already close to 0.8. To understand this effect, recall that GDSF evicts files using a key that combines size, frequency, and age of a file in the cache; larger, less-frequently used, and "older" files are more likely to be evicted from the cache when a new request arrives at the server. Thus, in the case of locality-disruption attacks, in which the attacker requests *new* unpopular files without building any file locality, the file-size part of the GDSF eviction key makes the algorithm vulnerable to locality-disruption attacks. In our experiments, we set the attackers' file size to 1 MByte. On the other side, the mean file size in the p2p workload is 5 MBytes. Hence, the attackers' files are less likely evicted from the cache, and pollution effects are magnified. We evaluate the file-size effects in more depth below.

On the other hand, Figure 5(a) indicates that LRU and LFU are more resilient to attacks, because the attackers are required to generate a larger number of

requests in order to increase the level of pollution. Nevertheless, the damage ratio can still be quite high. Moreover, since no counter-pollution mechanisms are currently implemented in Internet proxy caches, even higher-rate pollution attacks are quite feasible.

Figure 5(b) reveals somewhat different trends in the case of false-locality attacks. First, the GDSF's aging mechanism limits the damage ratio to 0.4 in this scenario. Because the attackers build a false locality by re-requesting the same set of files, the files' age increases over time, leading to their eviction from the cache. However, even a relatively moderate damage ratio can dramatically degrade the system's performance. For example, because p2p traffic accounts for the vast majority of the network's total bandwidth [6], a slight drop in hit ratio of a p2p cache will result in a large increase in network traffic. A simple calculation reveals that when the hit ratio is 0.6 in the absence of attack, a byte damage ratio of 0.4 can cause an additional *hundreds of TBytes* of data to be transferred over the access link on a daily basis. [5] Second, Figure 5(b) indicates that false-locality attacks more severely degrade the LFU algorithm. Indeed, as the number of malicious requests increases, the frequency count for falsely popular files increases, and the pollution becomes more significant.

3.3 System Factors

3.3.1 Attackers' file size

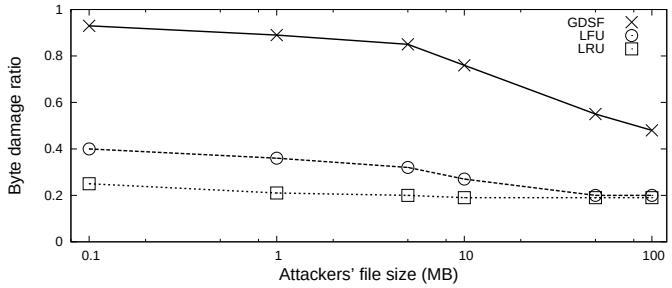
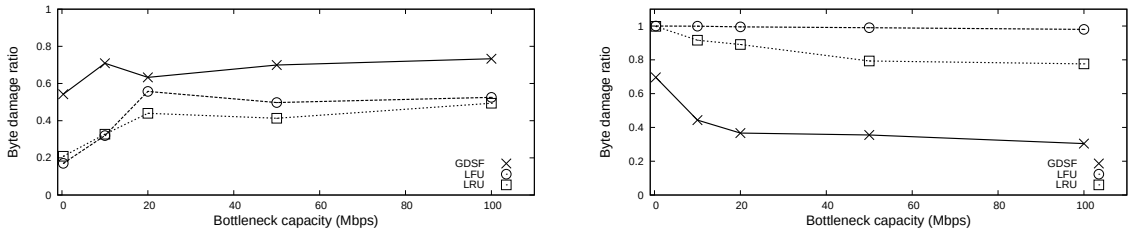


Fig. 6. The role of attackers' file size

The file size used by attackers influences the effectiveness of pollution attacks in a non-trivial manner. The above experiments indicate that the use of small files can be a particularly efficient attack strategy against certain replacement algorithms (*e.g.*, GDSF). Here, we explore such effects in more depth.

Figure 6 depicts the byte damage ratio as a function of the file size used in locality-disruption attack. The aggregate attackers' throughput (in bps) is

[5] Assume a nominal traffic load of 100 Mbps on the access link. The Byte damage ratio of 0.4 degrades the hit ratio from 0.6 to 0.36. This increases traffic by 24 Mbps, resulting in an increment of 260 TBytes daily.



(a) Locality-disruption attack (b) False-locality attack

Fig. 7. The role of the access link capacity

kept constant, approximately to 20% of the throughput achieved by regular clients. Despite the attackers' much lower request rate, the byte damage ratio is significantly larger in the case of GDSF, as explained above. As the attackers' file size increases, the damage ratio decreases. Indeed, the larger the attackers' files, the sooner they get evicted by GDSF, and the damage ratio becomes less pronounced. The knee in the GDSF curve arises around 5 MBytes, which corresponds to the mean file size for the p2p trace.

The attackers' file size affects the LFU and LRU algorithms less dramatically. First, the absolute byte damage ratio is much lower than in the GDSF case; second, as the attackers' file size increases, the byte damage ratio decreases for LFU. This is because we keep the throughput constant; thus, increasing the attackers' file size means decreasing the attackers' request frequency, which weakens the attack. Finally, the impact of the attackers' file size is smallest for LRU.

3.3.2 Access link capacity

A large discrepancy between local and access network capacities is one of the key reasons for implementing proxy caches: they reduce traffic on typically bandwidth-scarce access links. Below, we evaluate the role that this discrepancy can have on the effectiveness of pollution attacks.

In our experiments, we set the local capacity to 1 Gbps, and vary the access capacity from 300 kbps to 100 Mbps. At the same time, we control the number of requests generated by regular clients such that they consume 50% of the access bandwidth on average, whereas the rest is utilized by attackers. While not representative of an actual scenario, our main goal here is to illustrate the impact of limited access capacity on the effectiveness of the attack.

Figure 7(a) depicts the byte damage ratio as a function of the access link capacity in the presence of locality-disruption attacks. The key observation is that the effectiveness of the attack is reduced for lower access capacities. Because the locality-disruption attack considers downloading new unpopular files, the attackers have to *share* the access link with regular clients. Conse-

quently, when the access link capacity is the system bottleneck, the power of the attack is significantly reduced. For example, for LRU and LFU, the byte damage ratio is only 0.2 when the access link capacity is limited to 300 kbps. This doesn't hold for GDSF, which is vulnerable to low-rate pollution attacks. As the access link capacity increases, the attackers manage to make a larger number of requests, and hence the pollution becomes more pronounced for all replacement algorithms.

Figure 7(b) shows opposite effects for false-locality attacks: increasing the access capacity weakens the effectiveness of the attack. The key reason for such a behavior is the fact that malicious clients manage to build a false locality over longer time intervals despite limited access capacity. Once this is achieved, the access rate is no longer a limitation, and the attack becomes powerful. This holds particularly for LFU and LRU, where the byte damage ratio is approximately one for moderate link capacities. Again, the GDSF case is different, because this replacement mechanism limits the pollution level for false-locality attacks. Finally, as the access capacity increases, the effects of the attack slightly weaken. This is because we increase the request rate of regular clients, as explained above.

3.3.3 The Impact of the Zipf parameter α

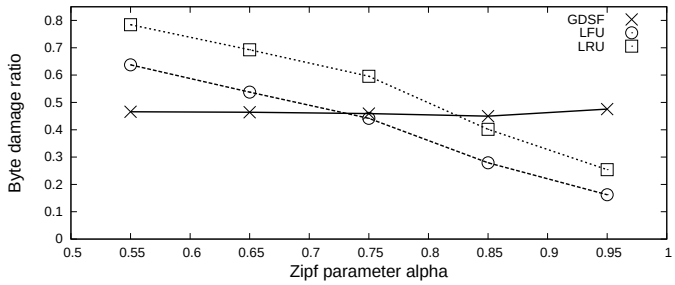


Fig. 8. The impact of the Zipf parameter α

We explore the impact of the Zipf parameter α, which we use to generate the web traces. Breslau *et al.* [35] showed that the distribution of page requests generally follows a Zipf-like distribution; the relative probability of a request for the i-th most popular page is proportional to $1/i^{\alpha}$, with α varying from trace to trace, ranging from 0.64 to 0.83.

Figure 8 shows the byte damage ratio as a function of α. For smaller α, the effects of the attack are more devastating, particularly for LFU and LRU. This is because the correlation structure of requests is "weaker" when α is small, such that the regular users do not manage to build strong file locality. As α increases, the requests become more and more correlated, such that the effects of the attack become less pronounced. On the contrary, the byte damage ratio is almost flat in the GDSF case. The key reason is GDSF's aging mechanism

which weakens the level of pollution. Unfortunately, the aging mechanism is not perfect: for larger α, it starts evicting *regular* popular files, such that the byte damage ratio becomes larger than it is in the cases of LRU and LFU.

4 Counter-Pollution Techniques

In this section, we examine how to effectively and efficiently detect the proposed cache pollution attacks and mitigate their effects.

The cache pollution attacks are hard to detect because all requested files are uncorrupted. Thus, one cannot base the detection scheme on the contents of the cached files. Another and even more serious challenge is the following. The traditional detection schemes usually observe the clients' access patterns and detect anomaly/attacks when such patterns change significantly. These schemes have to identify the clients by their IP addresses. However, many residential users (and even some business users) obtain their IP address dynamically through the DHCP protocol (especially in Asia). That is, for any client, the IP address it obtains may differ from one day to the other. Given very long time-scales of pollution attacks, it is very challenging to identify the history patterns for any of the clients. Consequently, to the best of our knowledge, no existing detection scheme applies to the proposed pollution attacks.

To address these challenges, we analyze the inherent characteristics of these attacks and design several schemes to detect false-locality attacks and locality-disruption attacks separately, initially proposed in [1]. We can further detect these two attacks even when they are combined and mixed together. Again, we focus on the attack detection in forward proxies, including pollution attacks against local DNS servers. Attacks against reverse (server-side) proxies, in general, are harder to achieve because such proxies are usually only responsible for the objects served by corresponding back-end servers. Thus, it is hard to upload unpopular files to such caches, especially for locality-disruption attacks, which *require* a large number of such files. However, false-locality attacks are still quite applicable. In addition, reverse proxies usually serve much more clients relative to forward proxies, hence more potential attackers. We will demonstrate that our proposed solutions, based on bloom filters and probabilistic counting, are highly scalable. As such, they are equally applicable to both types of proxies.

4.1 Detecting False Locality Attacks

In Section 3.2, we found that the false-locality attack is more effective than the locality-disruption attack, especially when the LFU replacement policy is used and/or when there is limited access link bandwidth connecting the cache server to the Internet. In this section, we attempt to detect such attacks.

The false-locality attack often involves distributed attackers making requests to the same set of files. Our first attempt is to use such a correlation to detect distributed attacks. That is, we look for clients that request a similar set of files that normally always reside in the cache. However, there are two heavy-tailed distributions in normal traffic patterns that hamper this method: 1) a small number of clients send a large number of requests; and 2) a small number of popular files are requested by a large number of clients. These two patterns will cause some clients to request a similar set of popular files which are always in the cache. These scenarios will raise false positives for correlation-based detection. These are validated with popular cache and Web server logs, *e.g.*, from MSNBC [36].

The above discussion inspires us to analyze the fundamental characteristics of false locality attacks: the repeated requests from the same IP to the unpopular files. Unless there are an extremely large number of attackers, they cannot keep the false locality without having each attacker making repeated requests for the same files. However, it is rare for a normal client to reload the same file multiple times in a short period, *e.g.*, a few hours or a day. As shown in several web/cache traces, the real popular files are accessed by a large number of regular clients (tens of thousands or more), and the number of such regular clients should be much larger than the number of attackers.

While some clients may keep loading certain search engines and news web sites, such dynamic content is uncacheable. Indeed, Bent *et al.* found that about 60% of HTTP requests are generated for dynamic content and are thus uncacheable [37]. The second caveat is that certain programs, *e.g.*, web crawlers, repeatedly request the same file until a successful download occurs. The cache server and our detection system can recognize such failed requests and exclude them from counting. Finally, it is also possible for some clients to solely keep loading the same web page, *e.g.*, "http://www.google.com," without placing any queries. However, there is only a small number of such popular web pages and the access patterns for those pages tend to be stable. Our approach to the problem is to create a "white list" of such pages, characterized by a large ratio of requests vs. the number of unique IPs that placed these requests.

Based on these observations, we design two detection mechanisms for false-locality attacks: one is to detect attackers each of whom has a large number

and percentage of repeated requests. The other is to detect the cached objects with false locality.

4.1.1 Attacker-based Detection

In this scheme, we record the files that each client requests over longer time scales, and calculate the following statistics: *(i) the number of repeated requests* and *(ii) the percent of repeated requests (ratio of repeated requests vs. the total request hits in the cache).* Only when both metrics exceed given thresholds, we mark such a client as the attacker. The goal is to avoid false positives while successfully detecting large pollution attacks. For example, some benign client IPs can generate a large number of requests, and consequently a relatively large number of repeated requests *e.g.*, due to NAT effects. However, the percent of repeated requests from the same IP is small in general. On the other hand, other clients could generate a small number of requests, but with relatively high percent of repeated requests (again, due to the NAT effects). Still, they are unlikely to be attackers. Also, note that for the second metric, we use "the number of total request hits in cache" as the denominator instead of "the number of total requests." This is to prevent an attacker from potentially evading the detection mechanism by launching false-locality and locality-disruption attacks simultaneously.

The key issue with this scheme is the memory consumption. We apply two methods to reduce it. First, we filter the requests by applying a threshold on the request rate before counting the repeated requests. Second, we use bloom filters to record the files requested over longer time scales, and significantly reduce the amount of states needed for counting repeated requests.

Scalable Detection with Bloom Filters. A bloom filter is a computationally efficient hash-based probabilistic scheme that can represent a set of *unordered* keys with minimal memory requirements, while answering membership queries with zero probability for false negatives and low probability for false positives [21]. In our case, the keys are requested URLs for web or requested files for p2p applications.

Given a Bloom filter m, an element e is inserted into m by hashing e with k different hash functions H_i, $i = 1...k$, and setting the corresponding bits, $m[H_i]$. Checking if an element belongs to the set involves hashing the element again as described above, and checking if all the corresponding bits are set. The accuracy depends on the size of m and k. In our case, even with 1 million attackers' files with false locality, we use 1B for each key such that the size of m becomes 1 MB. We set $k = 4$, while the percent of false positives for the membership test is 2%. Note that such an error rate does not directly translate to the detection error rate. In fact, our detection threshold is very insensitive

to such small errors as we will demonstrate below. In fact, we can tolerate even larger errors with less memory consumption. Assuming 100 attackers and another 100 normal clients making a large number of requests, currently the total memory consumption becomes 1MB×200 = 200MB.

4.1.2 Object-based Detection

Our second scheme is to detect objects with false locality. As discussed before, normally one client requests each file a small number of times to get the content. Thus the ratio of $number\ of\ requests$ vs. $number\ of\ unique\ IPs$ is small. However, for the false-locality attack, each attacker will request the same file multiple times to achieve false locality. Otherwise the attack will not be effective. So for the false-locality attack, the ratio of $number\ of\ requests$ vs. $number\ of\ unique\ IPs$ will be relatively large. Thus we design our detection scheme with the following two steps.

Step 1: Find cached files with locality. It is in fact very tricky to find cached files with locality. Usually, the effectiveness of a cache is measured by the hit ratio, *i.e.*, the number of hit requests divided by the total number of requests. However, it is hard to apply this metric directly to measuring the locality of each individual file because the technique strongly depends on the number of requests toward that file and the request patterns. Next, we considered adding the total hit/request number as a supplementary metric to measure the file locality. That is, a file is considered to have good locality when the hit ratio is larger than a certain threshold (*e.g.*, 90%) and the total request number is above a certain threshold (*e.g.*, 50 per hour). However, the threshold depends on the replacement policy and is difficult to set. For example, for LRU, even if the file has a large hit request number in the past minute, or even the past hour, it may be evicted in the next time interval when there is a large number of requests for different files; on the other hand, the file may have much better locality for LFU.

To overcome these limitations, we count the duration of a file's presence in the cache within certain time interval (*e.g.*, an hour) as its locality. We use two hash tables: one for counting the cache duration for each file, and the other for recording the most recent entry time for each file. For each file, we keep track of its cache duration and its last entry time. When a file is placed in the cache, we record its entry time. When it is evicted, we update the duration table by adding its duration within a predefined interval, *e.g.*, the most recent hourly interval (3pm - 4pm), and set its entry time to a special value, such as zero. At the end of each interval (*e.g.*, 4pm), we check for files that have been in the cache for certain percentage (*e.g.*, 80%) as files with locality. Meanwhile, we reset all the duration values to zero. Since we still keep the most recent entry time, if a file has been in the cache during the previous interval, we can

set its occupancy to 100%.

The state to maintain for each file is only 4B for the duration and 4B for the last entry time. However, we may need to maintain state for files that are not currently in the cache, and this can result in significant overhead over time. To reduce such overhead, at the end of each time interval, we remove entries of files that are not currently in the cache.

Step 2: Detect based on average requests per unique IP. After finding files with locality, we need to check if these are real popular files, or faked ones. As mentioned previously, we believe most users will request each file only once. Even if we consider the network address translator (NAT) effect, since most NAT users are consumer/home users, there is usually only a small number of clients behind each NAT. Thus for each file with locality, if the ratio between $number\ of\ requests$ and $number\ of\ unique\ IPs$ is larger than a certain threshold (*e.g.*, 4), we report this file is a false popular file and remove it from the cache.

The major challenge for this scheme is to count the unique number of IPs for each of the files with locality. Each popular file may be requested by hundreds of thousands of clients, so it is too costly to keep track of all of them. Since we need only an estimated value of the number of unique IPs, we leverage probabilistic counting [9,8], which we will discuss in the next section, to significantly improve scalability.

Scalable Detection with Probabilistic Counting. The problem of counting unique IPs can be abstracted as the problem of estimating the number of distinct elements n, *i.e.*, the cardinality, of a very large data set of size s. The traditional solution is to store everything in memory and sort data afterwards. Such algorithms have memory consumption in $O(n)$ and time consumption of $O(s \log n)$. Given that a large ISP of a class A network can have up to 2^{24} clients (*i.e.*, n), and that we need to count the number of unique IPs for every cached file with locality, such algorithms are clearly far too costly in terms of both storage consumption and processing time. In contrast, probabilistic counting provides a method to count the cardinality by using constant memory and doing only one pass on the data, *i.e.*, a runtime of $O(s)$. The tradeoff is that it never gives the exact cardinality, but rather an estimate with a certain precision.

Here we adopt a variant of probabilistic counting algorithm named *Probabilistic Counting with Stochastic Averaging* (PCSA, Figure 9) which employs *stochastic averaging* to increase the accuracy. When using 32-bit words for the hashed values, PCSA can count cardinalities well up to 10^8 with a standard error of 14% for a memory consumption of 128 bytes and 10% for 256 bytes of memory usage. Since we only need a rough estimate of the cardinality, we

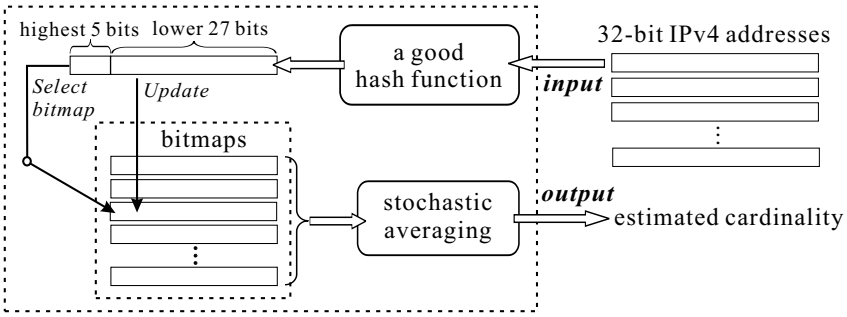


Fig. 9. Probabilistic Counting with Stochastic Averaging (PCSA)

choose the 128-byte PCSA for detection.

However, PCSA is not suitable for counting small cardinalities due to its inability to sense variations in this setting. From our experience, with 128-byte PCSA, when the actual cardinalities are less than 50, it gives extremely inaccurate estimates. Thus we use a hash table to record the IP addresses for each file first, and turn to PCSA counting only when the number of unique IPs exceeds 50. At that point, we simply add the existing 50 records to PCSA and remove the hash table. As a result, the maximum memory consumption for each cached object is $50 \times 4\text{B} = 200\text{B}$.

4.2 Detecting Locality Disruption Attacks

For locality disruption attacks, attackers keep requesting different unpopular files to destroy the locality of real popular files. There are two inherent symptoms for such attacks. First, the hit ratio is low. Second, the average life-time of all cached files is short. We design our detection scheme based on these two signatures. For each cached file, we record its entry time. Periodically, we compute the average durations for all files in the cache. When the average duration is very low, we detect the attacks. However, to *mitigate* such attacks, it is crucial to additionally detect the attackers. Thus, for each file in cache, we record the client IP making the most recent access request. When we detect the locality disruption attack, we check the IP addresses in the cache table and search for those that make most of the requests. Again, the assumption is that the number of attackers is much less than the number of normal clients [38]. Then, for locality disruption attacks to be successful, each attacker needs to request a relatively large number of unpopular files, and such amount is much larger than that of a normal client.

For example, for a cache of 100GB, even when the average file size is 100KB, it needs 1 million files to fill it. Thus, even for a distributed false-locality attack with 100 attackers, each attacker needs to have on average of 10,000 files loaded in the cache with its IP as a requestor. On the other hand, a typical

client will not *exclusively* request such a large number of files in the cache. Regular clients usually request popular files, and requests toward these files are often *shared* by many different clients. That is, when counting the number of requested files in the cache by the requestor IP, the attackers will surface with a large number of requested files in cache. This is precisely because the number of attackers is much lower than the number of regular clients.

4.3 Detecting Attack Combinations

Attackers can launch both false-locality attacks and locality-disruption attacks at the same time. For instance, the same set of attackers can launch false-locality attacks to occupy 50% of the cache space, and at the same time they start locality-disruption attacks to interfere the file locality in the rest 50% of the cache.

Such attacks are stealthier because the false-locality attacks will increase the average life time so that locality-disruption attacks may not be detected. The existence of locality-disruption attacks will *not* affect the detection of false-locality attacks discussed in Section 4.1. For attacker-based detection, both metrics need to be high enough for creating false locality even for 50% of the cache. Hence, our general detection strategy is as follows.

First, we try to detect false-locality attacks, and once detected, we exclude the files with false locality from calculating the average life time. Then the locality-disruption attacks can be detected separately.

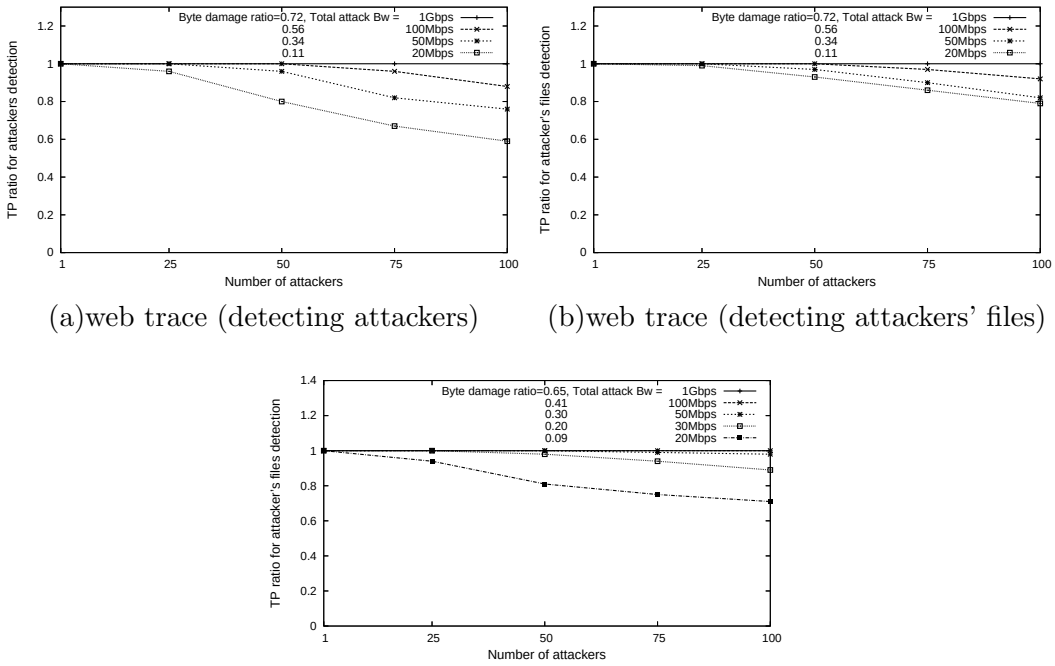
5 Pollution-Detection Evaluation

In this section, we use simulation to evaluate the effectiveness of the detection techniques discussed above.

5.1 Simulation Methodology

We use web and p2p traces as follows.

• Web trace: There are 10,000 regular clients and the cache size is set to 10GB. The average file size is 10KB. There are 5 million files accessed by regular clients, and 1 million attackers' files for false-locality attacks; also, there are 100 million attackers' files for locality-disruption attacks.



(a)web trace (detecting attackers) (b)web trace (detecting attackers' files)

(c)p2p trace (detecting attackers' files)

Fig. 10. The true positive (TP) ratio of attacker-based false locality attack detection

• P2p traces: There are 2,000 regular clients and the cache size is set to 10GB. The average file size is 10MB. There are 2,000 files accessed by regular clients, and 1,000 attackers' files in the case of false-locality attacks; also, there are 100,000 attackers' files for locality-disruption attacks.

Here, we choose 10GB cache size because a previous study shows that even one week of NLANR cache traces in 2002 has only 15.6GB unique bytes; moreover, the study shows that setting a cache size to be 30% of the total unique bytes provides close-to-optimal hit ratio [28]. We also expect similar simulation results to hold for larger cache sizes with larger network bandwidth.

Given that false-locality and locality-disruption attacks have different effects on the external access links, we have different bandwidth for normal clients and attackers for these two cases. For false-locality attacks, the total bandwidth for regular clients is 100Mbps, while the bandwidth for attackers changes from 20Mbps to 1Gbps. We assume that the external link is 100Mbps and the internal link capacity is 1Gbps. For false locality attacks, attackers' request rates are only limited by the internal link capacity. For locality-disruption attacks, since almost all the requests have to be fetched from external servers, the total bandwidth for regular clients and attackers is 100Mbps. And we assume that the internal link capacity is larger than that and thus is not a constraint. The rest of the attack parameters are the same as described in Section 3.1.

In our experiments, we consider the worst case, *i.e.*, the stealthiest attacks where attackers are well coordinated so that each of them simultaneously request different files. We further consider the network address translation (NAT) effect. Note that with NAT, there may be multiple clients requesting the same file from the same source IP address. Hence, this must be considered when calculating the repeated requests. It was reported that 17% to 25% of Internet access is through a NAT-enabled gateways [39] and we randomly assign 20% of the regular clients to be behind the NAT. Since most NAT users are consumer/home users, the multiplex rate is usually small. We assume that on average there are 10 clients sharing the same NAT and have the same IP.

The evaluation metrics include: *(i)* the true positive ratio, defined as the percentage of real attacks detected, *(ii)* the number of false positives, defined as mis-detected attacks, *(iii)* detection time, and *(iv)* the memory consumption. The evaluation results for web traces and p2p traces are very similar. Due to space limitations, we only demonstrate the attacker-based false locality detection results for web traces, and report all the results based on p2p traces.

5.2 Results for False-locality Attacks

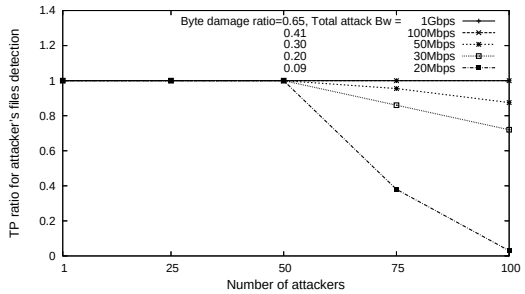
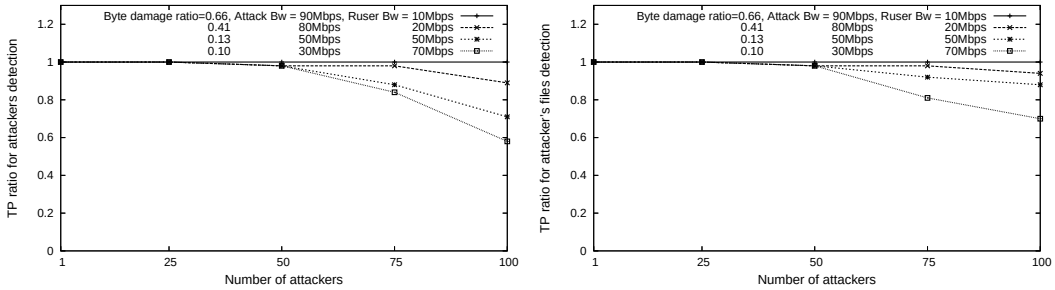


Fig. 11. The true positive (TP) ratio of object-based false locality attack detection



(a)Detecting attackers (b)Detecting attackers' files

Fig. 12. The true positive (TP) ratio of locality disruption attack detection. ("Ruser Bw" = regular users' bandwidth; "Attacker Bw" = attackers' bandwidth.)

1) Attacker-based Detection

Results for web traces. The first step for detection is to find suspicious clients which make a large number of requests. For the web trace simulation, we assume all normal users' traffic is web traffic. Given 10,000 regular users and a total bandwidth of 100Mbps, on average each normal user sends out 450 requests/hour. Considering the heavy-tail distribution of normal user request rates (see Section 3.1), we set the threshold as 5,000 requests/hour. This is 247 clients out of a total of 10,100 (normal clients plus attackers).

After the filtering, we set 5,000 as the threshold for *the number of repeated requests per hour*, and conservatively set 30% as the threshold for *the ratio of repeated requests vs. total requests hit in cache*. Figure 10(a) demonstrates the number of attackers detected by our scheme, while Figure 10(b) presents the number of attackers' files detected from these attackers. Because there is up to 80% fluctuation to the number of requests sent by each attacker, some of attackers may have a small request rate, smaller than the 5,000 threshold. Still, we can always detect those attackers with large sending rates, which is precisely why the result of Figure 10(b) is better than that of (a). Figure 10(b) also shows the power of mitigation once we block the traffic from the attackers. For all configurations, there is no false positive in these detections.

For attacks with different total bandwidth, the average detection varies from two to ten hours. The more total bandwidth an attack sends, the less detection time we need. Actually the mildest attack that we spend 10 hours to detect needs more than 20 hours to fully have the damage take into effect. But during this period, we already detect the attacks and mitigate them before they can fully pollute the cache.

In terms of memory consumption, when using bloom filter to record 1M different request files, as discussed in Section 4.1, we allocate 1MB bloom filter for each suspicious client, and only use a total memory of 1MB$\times$ 247 = 247MB. On the other hand, without bloom filters, we have to record all the files requested by suspicious clients. We represent each URL with a 16-byte MD5 hash of the URL to record them. Assuming that those heavy hitter clients on average make requests for 1M different files as each of the attackers does, the total memory consumption is 16B$\times$247$\times$ 1M = 3.952GB. When the number of suspicious clients and number of requested files increase, the detection system can easily run out of memory.

We implemented both methods and they achieve the same accuracy as shown in Figure 10. Since we only need a rough estimate on the number of repeated queries, the 2% false positive of bloom filters does not cause any detection inaccuracy.

Results for p2p traces. Because the average file size for p2p traces is 1,000

times larger than that of web traces, with the same bandwidth, the request rate from both attackers and normal clients are reduced proportionally. Thus we set the threshold of *request rate per hour* and *the number of repeated requests per hour* as 10, but leave the repeated request ratio metric unchanged as 30%. Again, we achieve accurate results as shown in Figure 10(c). There is only one false positive for each detection because the client is a heavy user who keeps loading a small set of popular files. In fact, such false positives can be filtered by applying the "white list" approach discussed in Section 4.1.

2) Object-based Detection

In this subsection, we evaluate the object-based detection and compare it with the attacker-based detection. Due to space constraints, we only show the results for p2p traces.

First, we set the duration threshold for a file to have good locality as 80% of one hour. In such a case, there are 813 files passing the locality test when total attack bandwidth is 1Gbps and the byte damage ratio is 0.65. With the decrease of attack rate and byte damage ratio, the number of attacker's unpopular files passing the locality test also decreases, *e.g.*, to 158 when the total attack bandwidth is 20Mbps, and the byte damage ratio becomes 0.08. Then we detect with the number of requests per unique IP. Given that 20% of IPs are behind a NAT, and on average 10 clients are behind each NAT, the *average* number of normal requests per unique IP is $0.8 \times 1 + 0.2 \times 10 = 2.8$. We conservatively set the detection ratio threshold to 4.

Figure 11 demonstrates the accuracy of our object-based detection scheme. When the byte damage ratio is greater than 0.3, our system can detect 88% of attackers' files within 10 hours. All detection results have zero false positives. The accuracy degrades as the byte damage ratio decreases. For attacks with small sending rates, the detection is less accurate than the attacker-based detection essentially because the repeated requests tend to be more dispersed and less obvious. For such light damage attacks, the longer the detection time, the better the detection results are.

To count the number of unique IPs, we implement both hash-table based accurate counting and PCSA to record the access IPs for each file with locality. Their detection results are exactly the same, but PCSA significantly reduces memory consumption. Given a 10GB cache, for web traces, the average file size is 10KB, there can be up to 1 million files with good locality. Even when each file is accessed by 10,000 clients (the maximum can be 2^{24} clients), we need 1M $\times$10000$\times$ 4B = 40GB of memory to record and update online. However, for PCSA-based recording, for each cached file, we either need to record the first 50 unique IPs (200B) or use PCSA for counting (128B). The memory consumption in the worst case is 1M$\times$200B = 200MB. In our detection

experiments of web traces, PCSA uses only about 2-5% of the memory as compared to that of the per-IP counting technique without noticeably sacrificing of detection accuracy.

5.3 Results for Locality-disruption Attacks

For locality-disruption attacks, the total bandwidth available to regular clients and attackers is 100Mbps. We vary the ratio of regular clients and attackers to obtain different byte-damage ratios as shown in Figure 12. The normal hit ratio varies from 0.5 to 0.6, and the average lifetime for those files in cache is about 5,000 seconds. Thus, to determine whether a locality disruption attack happens, we set the threshold for hit ratio to 0.3 and the threshold for average lifetime to 3,000 seconds. Once it occurs, we search for the IP addresses which account for the majority of the requests. On average, there are 1,000 files in the cache; given that the number of regular clients is approximately 2,000, each regular user has the hit rate of around 0.5. Still, given the heavy-tail distribution of normal client request rates, we set the threshold to 10; thus, when an IP address in the cache table appears more than 10 times, we report it as an attacker.

Figure 12 shows the number of detected attackers and blocked requests. Similarly to the false-locality detection results, the percent of blocked files is larger than the percent of blocked attackers. This is because all high-rate attackers are successfully detected and thwarted. For example, even when the byte damage ratio is as small as 0.10, the proposed scheme can detect more than 88% of unpopular file requests.

5.4 Results for Combination Attacks

We also simulate the detection for the combination attacks described in Section 4.3. In addition to varying the attacker and normal user total bandwidth as we did in the locality-disruption attacks, we vary the attacker bandwidth allocation to false-locality vs. locality-disruption attacks as shown in Figure 13. We obtain accurate results with no false positives. It demonstrates that our detection schemes are resilient to the mixture of attacks.

6 A Prototype Implementation

We implement and test our counter-pollution mechanisms by upgrading a Squid 2.5.11 caching server [27]. A detailed description of our implementation

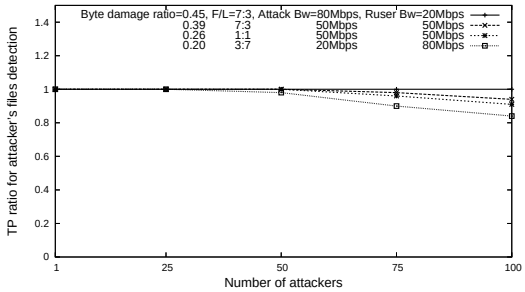


Fig. 13. The true positive (TP) ratio of combination attack detection.

and testing efforts is provided in [40]. Below, we briefly highlight the main components of our solution.

6.1 Anti-pollution Engine System

Figure 14 depicts the detection mechanism, which we implement by developing an add-on program called Anti-pollution Engine (AE) system.

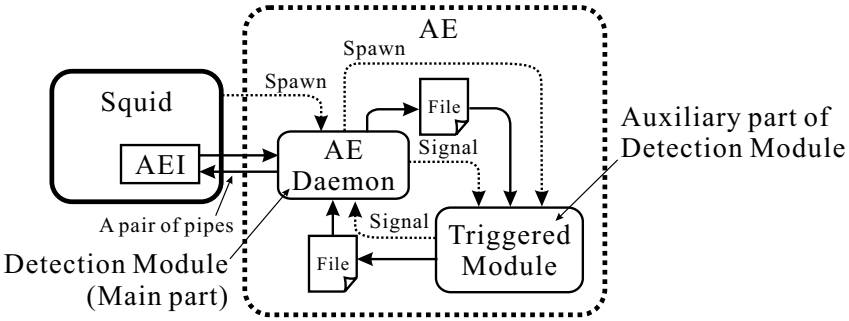


Fig. 14. Anti-pollution Engine system

The system consists of two parts: the AE and the AE Interface (AEI). The core part of the AE is the AE Daemon. It communicates with the AEI through a pair of pipes. The AE Daemon operates in the blocking mode for both pipe reading and pipe writing, while AEI operates in the non-blocking mode. AEI intercepts access information such as the client IP, requested URL, object size, reference count from Squid, and sends it to the AE Daemon. If the Detection Module infers polluted objects, the AE Daemon issues "block-entry" commands to the AEI through the pipe. Likewise, if the attackers' IP addresses are detected, the AE Daemon issues "block-client" commands to the AEI. Upon receiving "block-entry" or "block-client" commands, the AEI executes corresponding operations to counter the attack.

The main part of the Detection Module is embedded directly into the AE Daemon. In addition, the AE Daemon spawns one or more Triggered Modules at startup and sends signals to Triggered Modules to trigger operations whenever necessary. A Triggered Module is designed to run the auxiliary part of the

Detection Module in parallel with the AE Daemon. Upon receiving a signal from the AE Daemon, a Triggered Module reads data from a file generated by the AE Daemon, executes auxiliary operations and outputs results to another file. When exiting, it sends back a signal to the AE Daemon which in turn reads the output file and performs corresponding operations.

The AE Daemon is implemented as a process spawned by Squid at startup and the AEI is a module directly embedded in the Squid. The pair of pipes between AE Daemon and AEI can be easily replaced with a TCP/UDP socket, such that the AE can run on a separate machine.

6.2 Experiment Methodology

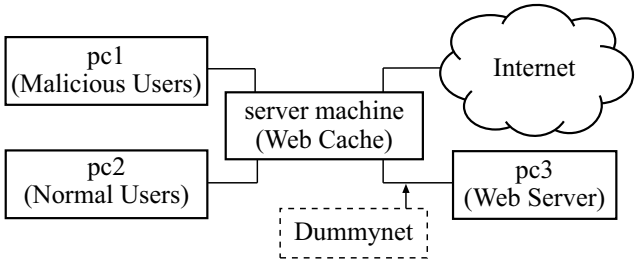


Fig. 15. Testbed topology

Figure 15 depicts the testbed, which consists of three PCs and one server machine running FreeBSD 5.3. We develop a simple tool to generate HTTP requests based on trace files. To simulate requests from different source IPs, we piggyback IP addresses in the URL part of requests and add corresponding codes into Squid to extract and remove the piggybacked IPs before Squid handles the requests. The web cache serves web requests both from regular and malicious clients. If a regular request cannot be served by the cache, the cache forwards the request to the origin server on the Internet. When a malicious request cannot be served by the cache, the request is forwarded to the web server, which hosts unpopular files used in the attack. The web server runs Apache software [41]; in addition, it runs Dummynet [42], an IP-layer network emulation package which mimics WAN bandwidth variations on the link between the web server and the cache.

6.3 Experiment Results

The results from the testbed experiments line up well with simulations. To demonstrate the basic system functionality, we show a sample result below.

Figure 16 plots the hit ratio of normal users in presence of a false-locality

Number of normal objects	20000
Average size of normal objects	14 kB
Number of polluted objects	70
Average size of polluted objects	2 MB
$\frac{Aggregate\ request\ rate\ of\ attackers}{Aggregate\ request\ rate\ of\ normal\ users}$	$\frac{2}{5}$
Cache size	100 MB(Squid's default)
Replacement algorithm	LRU

Table 3
Experiment parameters

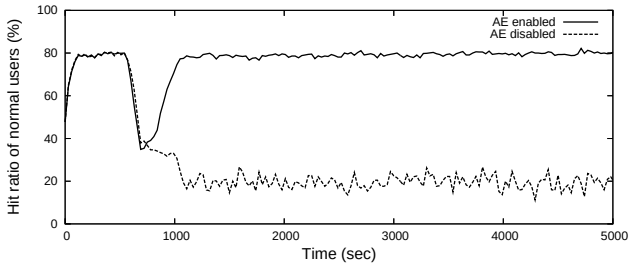


Fig. 16. Experimental results

attack. Parameters related to the traces and settings of the web cache are listed in Table 3. To achieve the steady state operating point, we let the system "warm up" such that the normal users experience high hit ratio, around 80% in this scenario. After about 100,000 normal requests, we launch the attack. In absence of any protection, the hit ratio of normal users drops quickly to approximately 20%. When the AE is enabled, it detects and blocks polluted objects, thereby restoring the hit ratio back to 80% in a short time period.

7 Related work

A class of file-level pollution attacks arises in p2p networks. For example, injecting a massive number of corrupted files into a p2p network is used as a common technique to decrease the availability of a specific item (*e.g.*, movie or software distribution) [43,44,14]. The key difference between p2p- and proxy-cache-targeted pollution attacks is that in the latter case, the attackers pollute the cache with *unpopular*, rather than bogus files, making them much harder to detect. In addition, the proxy caching pollution problem can affect a much broader Internet community, not only p2p networks.

The problem of *unintentional* caching pollution has been addressed in [26]. The authors find that referencing patterns of robots and crawlers, deployed

by search engines, can disrupt the cache file locality and increase the miss ratio of *server-side* web caches. Similarly, our locality-disruption attacks have a referencing pattern that intentionally disrupts the file locality of proxy caching servers. Also, unintentional caching pollution can arise during flash-crowd events. Thus, some cache replacement algorithms apply aging policies to discriminate against *old* popular objects [29,30]. Unfortunately, our preliminary results clearly indicate that such mechanisms are unable to protect the cache against intentional pollution attacks.

Malicious clients can send a large number of requests to pollute a proxy server. While it may appear possible to simply re-apply solutions designed to protect web servers from DoS attacks (*e.g.*, [45,46]), this is not the case. The key reason is a large gap in request rates between web-server- and proxy-cache-targeted attacks; cache-targeted attacks operate on longer time-scales and can pollute the cache without exhausting any proxy-server's resource. Thus, such attacks are simply invisible to the state-of-the-art web-server protection mechanisms.

Precisely due to crucial importance of the DNS infrastructure for the web, it is often a target of malicious clients [47–50]. In such scenarios, the attackers choose to flood the root or top-level domain name servers attempting to deny service to its clients. Our research differs in that it proposes a new class of pollution attacks targeted against the low-level DNS server's caching mechanism. Note that low-level DNS servers do have a default mechanism which limits the number of recursive queries within an interval it will support. However, its purpose is to minimize the recursive state the server has to keep while it fetches answers. As such, it is far from being capable of solving the problem of low-rate cache-targeted pollution attacks.

8 Conclusions

Internet caching servers are providing a tremendous service to the entire Internet community. In this paper, we argued that the lack of cooperation among proxy caches makes them vulnerable to a class of pollution attacks. We proposed and evaluated two such attacks: locality-disruption and false-locality attacks. Using representative web and p2p workloads, we showed that there exists a high variability in resiliency to pollution of investigated replacement algorithms (LRU, LFU, and GDSF). Yet, we demonstrated that replacement algorithms alone are fundamentally limited in their ability to protect the system against attacks. We developed, implemented, and evaluated a set of scalable counter-pollution mechanisms based on streaming computation techniques. Due to the high detection accuracy of the proposed solutions, attackers must launch highly distributed attacks in order to elude detection, which strongly removes incentives for conducting such transgressions.

References

[1] Y. Gao, L. Deng, A. Kuzmanovic, Y. Chen, Internet cache pollution attacks and countermeasures, in: Proceedings of IEEE ICNP, Santa Barbara, California, USA, 2006.

[2] V. Pai, L. Wang, K. Park, R. Pang, L. Peterson, The dark side of the Web: An open proxy's view, in: HotNets, 2003.

[3] Akamai, http://www.akamai.com/.

[4] O. Saleh, M. Hefeeda, Modeling and caching of peer-to-peer traffic, in: IEEE ICNP, 2006.

[5] CNET News.com, P2p caching: Unsafe at any speed?, 2003, http://news.com.com/2100-1025_3-1027508.html.

[6] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, J. Zahorjan, Measurement, modeling, and analysis of a peer-to-peer file-sharing workload, in: ACM SOSP, 2003.

[7] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of ACM 13 (7) (1970) 422–426.

[8] P. Flajolet, G. N. Martin, Abc, in: the 24th Annual Symposium on Foundations of Computer Science, 1983, IEEE Computer Society Press.

[9] P. Flajolet, G. N. Martin, Probabilistic counting algorithms for data base applications, Journal of Computer and System Sciences 31 (2) (1985) 182–209. URL `citeseer.ist.psu.edu/flajolet85probabilistic.html`

[10] J. Jung, B. Krishnamurthy, M. Rabinovich, Flash crowds and denial of service attacks: Characterization and implications for CDNs and Web sites, in: ACM WWW, 2002.

[11] D. Wessels, Report on the effect of the independent council report on the NLANR Web caches, http://www.ircache.net/Statistics/ICreport/ (1998).

[12] Australian IT, Music industry raids KaZaA offices., 2004, http://www.afterdown.com/news/archieve/4948.cfm.

[13] BBC News, File swappers fight back, 2003, http://news.bbc.co.uk/1/hi/technology/3013065.stm.

[14] J. Liang, R. Kumar, Y. Xi, K. Ross, Pollution in p2p file sharing systems, in: IEEE INFOCOM, 2005.

[15] J. Jung, E. Sit, H. Balakrishnan, R. Morris, DNS performance and the effectiveness of caching, IEEE/ACM Transactions on Networking 10 (5) (2002) 589–603.

[16] Verisign. the domain name industry brief, http://www.verisign.com/Resources/Naming_Services_Resources/Domain_Name_Industry_Brief/ (Nov. 2005).

[17] D. Bernstein, How to adjust the cache size, `http://cr.yp.to/djb.html`.

[18] M. Handley, A. Greenhalgh, The case for pushing DNS, in: Proceedings of HotNets, College Park, Maryland, 2005.

[19] J. Wang, X. Liu, A. Chien, Empirical study of tolerating denial-of-service attacks with a proxy network, in: Usenix Security Symposium, 2005.

[20] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, K. Worrell, A hierarchical Internet object cache, in: USENIX Technical Conference, 1996.

[21] L. Fan, P. Cao, J. Almeida, A. Broder, Summary cache: A scalable wide-area Web cache sharing protocol, in: ACM SIGCOMM, 1998.

[22] M. Rabinovich, J. Chase, S. Gadde, Not all hits are created equal: Cooperative proxy caching over a wide area network, in: WWW Caching Workshop, 1998.

[23] J. Touch, The LSAM proxy cache - a multicast distributed virtual cache, in: WWW Caching Workshop, 1998.

[24] R. Tewksbury, Is the Internet heading for a cache crunch?, 1998, http://www.isoc.org/oti/printversions/0198prtewks.html.

[25] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, On the scale and performance of cooperative Web proxy caching, in: ACM SOSP, 1999.

[26] V. Almeida, D. Menasci, R. Riedi, F. Ribeiro, R. Fonseca, W. Meira, Analyzing robot behavior and their impact on caching, in: Workshop on Web Caching and Content Delivery, 2001.

[27] Squid Web proxy cache, http://www.squid-cache.org/.

[28] A. Balamash, M. Krunz, An overview of Web caching replacement algorithms, IEEE Communications Surveys & Tutorials 6 (2) (2004) 44–56.

[29] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, T. Jin, Evaluating content management techniques for web proxy caches, in: HP Labs TR, 1999.
URL `http://www.hpl.hp.com/techreports/1999/HPL-1999-69.html`

[30] J. Dilley, M. Arlitt, S. Perret, Enhancement and validation of Squid's cache replacement policy, in: HP Labs TR, 1999.
URL `http://www.hpl.hp.com/techreports/1999/HPL-1999-69.html`

[31] F. Smith, F. Campos, K. Jeffay, D. Ott, What TCP/IP protocol headers can tell us about the Web, in: ACM SIGMETRICS, 2001.

[32] Y. Chen, L. Qiu, W. Chen, L. Nguyen, R. Katz, Efficient and adaptive Web replication using content clustering, IEEE Journal on Selected Areas in Communications 21 (6) (2003) 979–994.

[33] L. Qiu, V. Padmanabhan, G. Voelker, On the placement of Web server replica, in: IEEE INFOCOM, 2001.

[34] S. Sen, J. Wang, Analyzing peer-to-peer traffic across large networks, IEEE/ACM Transactions on Networking 12 (2) (2005) 219–232.

[35] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: Evidence and implications, in: IEEE INFOCOM, 1999.

[36] MSNBC, http://www.msnbc.com.

[37] L. Bent, M. Rabinovich, G. M. Voelker, Z. Xiao, Characterization of a large web site population with implications for content delivery, in: International World Wide Web Conference, 2004.

[38] M. Walfish, H. Balakrishnan, D. Karger, S. Schenker, DoS: Fighting fire with fire, in: HotNets, 2005.

[39] G. Armitage, Inferring the extent of network address port translation at public/private internet boundaries, Tech. Rep. CAIA TR 020712A (2002).

[40] Anti-pollution engine system, http://tough.cs.northwestern.edu/AE/.

[41] The Apache Software Foundation, http://www.apache.org/.

[42] Dummynet, http://info.iet.unipi.it/~luigi/ip_dummynet/.

[43] N. Christin, A. Weigend, J. Chuang, Content availability, pollution and poisoning in peer-to-peer file sharing networks, in: ACM E-Commerce Conference, 2005.

[44] D. Dumitriu, E. Knightly, A. Kuzmanovic, I. Stoica, W. Zwaenepoel, Denial of service resilience in peer-to-peer file sharing systems, in: ACM SIGMETRICS, 2005.

[45] S. Kandula, D. Katabi, M. Jacob, A. Burger, Botz-4-Sale: Surviving DDoS attacks that mimic flash crowds, in: ACM NSDI, 2005.

[46] W. Morein, A. Stavrou, D. Cook, A. Keromytis, V. Misra, D. Rubenstein, Using graphic turing tests to counter automated DDoS attacks against Web servers, in: CCS, 2003.

[47] k. claffy, Nameserver DoS attack, `http://www.caida.org/projects/dns-analysis/oct02dos.xml` (Oct. 2002).

[48] D. Moore, G. Voelker, S. Savage, Inferring Internet denial-of-service activity, in: Proceedings of Usenix Security Symposium, Washington, DC, 2001.

[49] N. Brownlee, k. claffy, E. Nemeth, DNS measurement at a root server, in: Proceedings of Globecom, San Antonio, TX, 2001.

[50] J. Hu, Blackout hits major web sites, `http://news.com.com/Blackout+hits+major+Web+sites/2100-1038_3-5234500.html`.