POSTER: Cashing in on the File-System Cache

Trishita Tiwari
Boston University
Boston, Massachusetts
trtiwari@bu.edu

Ari Trachtenberg
Boston University
Boston, Massachusetts
trachten@bu.edu

ABSTRACT

We consider the disk cache (file-system cache) information channel, and show how it can be exploited on various systems to yield potentially sensitive information. Our approach can be used locally by an unprivileged adversary to detect whether another user is writing to disk, and if so, the rate at which data is being written. Further, we also show how an attacker can detect whether specific files have been recently accessed by the victim. We then extend this attack to remote access through a web server, using timing analysis to identify recent access of chosen pages.

CCS CONCEPTS

• **Security and privacy → Systems security**;

KEYWORDS

File System Caches, Side Channels, Covert Channels

ACM Reference Format:
Trishita Tiwari and Ari Trachtenberg. 2018. POSTER: Cashing in on the File-System Cache. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), Oct. 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3243734.3278499

1 INTRODUCTION

On many systems, disk input/output is slow, and file system caches are thus employed as an optimization. However, like last-level caches for memory, file system caches are typically shared amongst all users on the same machine. This opens the caches to potential timing attacks based on cache hits, misses, and flushes.

More precisely, when a user writes to a file descriptor, the corresponding data is written to the file system cache first. When the cache is full, it is flushed and the data is written to disk. We show that this periodic flushing allows a collocated (but unprivileged) attacker to infer whether another user is writing to disk at a given point in time, and, if so, the write-rate of the victim.

Further, due to slow disk access times, the latency difference of a cache hit versus a cache miss is clearly demarcated. This allows us to identify cache hits and misses both locally and remotely over the network, thereby allowing us to determine if specific files have been recently accessed by some user (local or remote).

Our contributions thus involve two types of exploits of the heretofore under-explored disk cache information leakage:

- Collocated attacks:
  (1) Detect if another user is writing to disk.
  (2) If a user is writing to disk, infer the user's write rate.
  (3) Infer if a specific file has been recently accessed.
- Remote Attacks:
  (1) Infer if a specific file has been recently accessed.

2 MOTIVATION

The types of disk cache information leakage we describe could lead to more significant attacks. We show initial results for some of these attacks, while the others are extensions or works in progress:

(1) Detecting access frequency for specialized web pages, such as admin log pages, could allow an attacker to infer admin operational patterns for further social-engineering or opportunity attacks.
(2) Disk cache hits and misses could also be used to encode information to form a covert channel. Further, since the distinction between a file system cache hit and miss is fairly conspicuous, such a covert channel could be used to transmit information even remotely over the network. This is a promising avenue that we are currently looking into.
(3) Detecting if certain system libraries have been recently accessed can link inputs to executions. Indeed, whereas Linux does reveal process names through the `ps` command, unprivileged adversaries are not given access to information about which libraries are used by these processes. Such information could reveal interesting behavioral information of processes, such as whether particular (vulnerable) cryptographic library has been recently accessed.
(4) Access patterns for files that are modified only on system or software upgrades could allow attackers to remotely identify the current version or upgrade frequency of various system components.
(5) Being able to infer write rates to log files could allow correlation between activity and error logs, lengths of log messages, and the like. Identifying disk writes could also reveal when file uploads happen on web-servers, database servers, and other systems.

3 RELATED WORK

There has been much work done in the field of memory-based cache side channel attacks. For instance, the FLUSH+RELOAD [7] attack shows how an adversary can extract encryption keys from GnuPG by timing accesses to certain instructions in shared last-level caches on both the same operating system and across collocated Virtual Machines. Within the realm of disks, there has been work that utilizes the contention that arises from sharing hard disks to create a covert channel between Virtual Machines [4]. Further, lazy

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3278499

page mappings have also been shown as a potential covert channel, since reading a page that has already been mapped into memory takes lesser time than processing the page fault that would result otherwise [5]. However, despite the fact that disks based channels are not uncharted territory, to the best of our knowledge, there is no work yet that utilizes the *caches* associated with them as a potential source of information.

4 BACKGROUND

4.1 Stat

The `stat` [3] system call in Linux provides, among other things, information on the most recent `atime` (access time) and `mtime` (modification time) of any given file, updates to which are triggered respectively by `read` and `write` operations to said file. Since kernel 2.5.48, `stat` supports nanosecond resolution for the file time-stamp fields [3]. However, since providing nanosecond resolution would lead to performance overhead, in reality, most implementations employ some optimizations including buffered writes [2], updating timestamps only under specific circumstances [1], and the like.

4.2 File System Caches

In order to reduce disk write latencies, most write operations performed on file descriptors write to the file system buffer first, and then, at some later time (or when the buffer is full), flush it and perform actual writes to disk.

Similarly, reading data from file descriptors stores the `inode` and the file data into the file system cache, and hence subsequent accesses to the same file are much faster. [6]

Operating systems today employ file system caches by allocating unused memory for such disk caching purposes. This essentially means that a cache hit is a memory read while a cache miss is a disk read, and the difference in the respective latencies is conspicuous enough to be detected locally, and even over the network.

5 COLLOCATED ATTACKS

Threat Model. For this attack, we assume that the attacker is an unprivileged adversary on the same machine as the victim. The victim machine uses the Linux kernel and therefore can `stat` files.

Stat Attacks. The attacker writes to his own file in a loop (at a known rate), and continuously runs `stat` on his file to monitor the `mtime` time-stamp. The attacker then obtains a time-series of file modification times for his own file, and then obtains a first order difference.

If the attacker has a large write rate (say, 10^6Bytes/Loop), then the disk cache fills up quickly and is forced to flush (having a smaller write rate will only prolong cache flushing, not obviate it – hence, the side channel will still exist). This shows up as both small and large peaks in Figure 1 (the small peaks are slightly greater than 0.003s, and therefore are not easily visible). Since the disk cache is shared, we see more frequent flushing than usual if there is another user who is simultaneously writing to his own file (see Figure 2).

This increase in the frequency of the peaks when another user is writing to disk with the attacker allows us to detect the victim's presence. Further, we could attempt to correlate the average time between consecutive peaks with the victim's write rate and see if

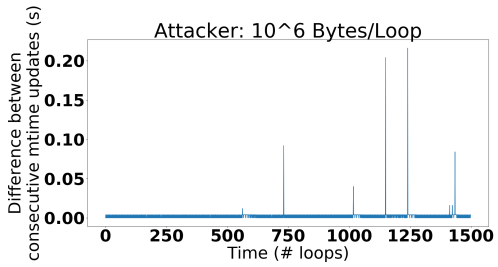

Figure 1: Consecutive modification time changes of the attacker's file when a write load is applied by just the attacker

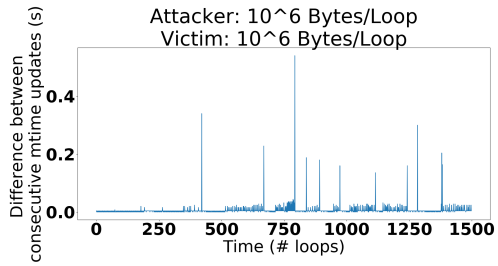

Figure 2: Consecutive modification time changes of the attacker's file when a write load is applied by both the attacker and the victim.

there is a discernible trend. (Peaks were detected by a simple threshold of 0.003s determined after manual inspection.) The resulting graph is shown in Figure 3. There are 10,000 data points, and each data point is an average of 3 trials. Here, we see the contours of a trend that reinforces the notion that a shorter period for flushing the disk cache means that the victim is writing at a faster rate, since the cache fills up faster. Indeed, as outlined in Section 2, such knowledge of write rates could be used to infer things like the length of log messages written, detect file uploads to web-servers or data written to database servers, and other such interesting information. However, we do note that the trend is noisy and warrants further investigation.

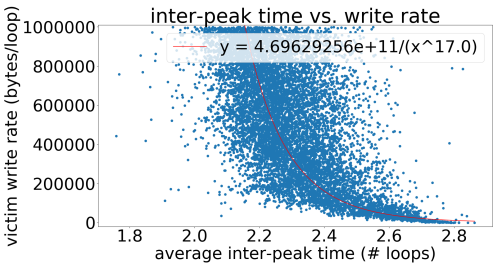

Figure 3: Victim's write rate vs. the average time between consecutive write latency peaks

Cache hits vs. Misses. Another attack involves exploiting timing differences between cache hits and misses. This could be used for a variety of purposes, one of which is to enumerate which libraries

the victim's system is using. If a user is already using a particular library, it should be present in the disk cache and therefore should load faster when the attacker attempts to access it.

For this attack, we have an experiment that involved repeatedly loading a python library (the `socket` library) under two conditions:

(1) When it was already loaded by another program before, triggering a cache hit.
(2) Loading the library when no one else used it, triggering a cache miss.

We conducted 3 trials, repeating both scenarios 2000 times per trial. The average latency for loading a library in the cache hit situation was 0.011s, 0.012s and 0.011s. However, in the cache miss scenario, the average load time was measurably higher: 0.030s, 0.032s and 0.029s (roughly 3x higher in each case). Further, the 99th percentile of the cache hit latency distribution was found to be at the 0th percentile of the cache miss scenario in all three trials, suggesting that cache misses are almost always larger than cache hits.

6 REMOTE ATTACKS

Threat Model. Here we assume that the attacker does not have physical access to the victim machine, but can access it over the network (i.e., the victim machine could be running a web-server).

Experimental Setup. Here, the victim machine was operating as a web-server that hosted a file named `index.html` which was accessible by any user. The web-server and attacker machines were on separate networks (but within the same geographical area).

Cache hits vs. Misses. To check if disk cache hits and misses are detectable over the network, we remotely accessed `index.html` 1813 times (without clearing the disk cache on the web server, thereby ensuring cache hits. We also ensured that there was no other intermediary network/proxy caching by setting the appropriate HTTP headers on `index.html`).

The mean access time of this distribution was 0.008s.

On the flip side, if the file is being accessed directly from disk (i.e., the file is not in cache), then the remote access times are higher.

To measure this, we repeated the previous experiment, only this time we manually cleared the disk cache on the web-server after every access to ensure cache misses.

The resulting distribution was composed of 1812 points (we removed 1 outlier), with a significantly higher mean access time of 0.023s.

Comparing both distributions, we see that the mean of the cache miss latencies is 2.875 times that of the cache hit access times, despite accounting for network transmission times. This lends credence to the inference that file system cache hits and misses can be identified over the network. This is further corroborated by the observation that 80% of the data points in the cache hit latency distribution lie below 0.009s, while only 36% of those in the cache miss access time distribution lie below said threshold. This suggests that even if we demarcate cache misses from hits by merely setting a hard threshold of 0.009s, we would be making correct predictions most of the time. Hence, this could be relatively reliably used to determine over the network if a certain file was recently accessed. Indeed, as discussed in Section 2, this technique could be used to obtain patterns on accesses of specialized web pages such as admin login pages, or could be used to create a covert channel to transmit information over the network solely through cache hits and misses.

7 CONCLUSIONS AND FUTURE WORK

We show how to potentially infer a variety of revealing information about a system by utilizing the timing information of reads, writes, and file accesses.

For the write rate detection attacks, future work would involve attempting to obtain a less noisy trend on Figure 3. The noise could be explained by optimizations on `stat` that preclude nanosecond precision. Or it could be due to our simplifying assumption that a peak is any latency above a hard threshold – clearly, not all peaks are of the same height (due to different layers of write-buffering).

For the file access detection attack, while we show that it is possible to discern (both locally and remotely) whether a file has been recently accessed, the ability to infer its access patterns over a long period of time would necessitate repeated measurements of the file's access latency. However, a potential issue with this attack is that an attempt by the attacker to measure access latency will itself involve accessing the file, thereby tainting the cache with said file's presence. This precludes any future measurements of this sort, as long as the file is still present in the cache. For our experiments, we manually cleared the cache between individual measurements (which requires super user privileges), but for a realistic attack scenario, there is need for an unprivileged/remote mechanism to accomplish cache flushing.

This will allow us to potentially craft an attack like FLUSH+RELOAD [7], where the attacker can time access to a certain file to see if its been recently accessed, flush the cache, wait for the victim to access the file, then repeat the process over a long period to obtain fine grained access patterns. Or we could potentially use a cache hit/miss to encode one bit, thereby allowing two (physically) separated colluding adversaries to communicate via a covert channel. The next step would therefore involve implementing a remote/unprivileged flushing mechanism to remove from the cache all traces of the file in question.

8 ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1563753. It is also supported in part by Boston University's Distinguished Summer Research Fellowship, and the department of Electrical and Computer Engineering.

REFERENCES

[1] Jonathan Corbet. 2009. That massive filesystem thread. https://lwn.net/Articles/326471/
[2] Jonathan Corbet. 2014. Introducing lazytime. https://lwn.net/Articles/621046/
[3] Michael Kerrisk. 2017. *stat(2) Linux User's Manual* (4.16 ed.). Linux man-pages project, http://man7.org/linux/man-pages/man2/stat.2.html.
[4] Bartosz Lipinski, Wojciech Mazurczyk, and Krzysztof Szczypiorski. 2014. Improving hard disk contention-based covert channel in cloud computing. In *Security and Privacy Workshops (SPW), 2014 IEEE*. IEEE, 100–107.
[5] Colin Percival. 2005. Cache missing for fun and profit.
[6] David A. Rusling. 1997. The VFS Inode Cache. http://www.science.unitn.it/~fiorella/guidelinux/tlk/node110.html
[7] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*, Vol. 1. 22–25.