usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

DupeFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels

Andrei Bacs and Saidgani Musaev, *VUSec, Vrije Universiteit Amsterdam;*
Kaveh Razavi, *ETH Zurich;* Cristiano Giuffrida and Herbert Bos,
VUSec, Vrije Universiteit Amsterdam

https://www.usenix.org/conference/fast22/presentation/bacs

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

DupeFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels

Andrei Bacs[†] Saidgani Musaev[†] Kaveh Razavi[‡] Cristiano Giuffrida[†] Herbert Bos[†]

[†] *VUSec, Vrije Universiteit Amsterdam* [‡] *ETH Zurich*

Abstract

To reduce the storage footprint with increasing data volumes, modern filesystems internally use *deduplication* to store a single copy of a data deduplication record, even if it is used by multiple files. Unfortunately, its implementation in today's advanced filesystems such as ZFS and Btrfs yields timing side channels that can reveal whether a chunk of data has been deduplicated. In this paper, we present the DupeFS class of attacks to show that such side channels pose an unexpected security threat. In contrast to memory deduplication attacks, filesystem accesses are performed asynchronously to improve performance, which masks any potential signal due to deduplication. To complicate matters further, filesystem deduplication is often performed at large granularities, complicating high-entropy information leakage. To address these challenges, DupeFS relies on carefully-crafted read/write operations that show exploitation is not only feasible, but that the signal can be amplified to mount byte-granular attacks *over the network*. We show attackers can leak sensitive data at the rate of ∼1.5 bytes per hour in a end-to-end remote attack, to leak a long-lived (critical) OAuth access token from the access log file of the nginx web server running on ZFS/HDD. Finally, we propose mitigations where read/write operations exhibit the same time-domain behavior, irrespective of the pre-existence of the data handled during the operation.

1 Introduction

Modern filesystems such as ZFS [9] and Btrfs [47] rely on deduplication to reduce the storage footprint for achieving scalable storage systems [17, 38, 59]. The idea is both simple and attractive: if two files both contain some data that is exactly the same, we can save storage space by storing the corresponding data once and maintaining shared references for the two files. Superficially, such functionality resembles its memory deduplication counterpart, where operating systems and hypervisors reduce the memory footprint by merging pages with the same content into a single shared copy-on-write (COW) page. Unfortunately, memory deduplication presents security risks as researchers have shown that it is possible to leak even high-entropy data by detecting when memory is shared [7, 10, 22]. In response, cloud providers and operating system vendors have simply disabled memory deduplication to stop these attacks [10, 29]. In contrast, filesystem deduplication is still commonly deployed everywhere [39, 55]. The question we ask in this paper is whether similar or even more significant security risks exist for filesystem deduplication.

At first sight, the answer appears to be an easy *no*. After all, memory and filesystem deduplication may have the same high-level objective and modus operandi, but their behavior is fundamentally different. In particular, filesystem operations tend to be asynchronous for efficiency. As an example, a write to a file, regardless of deduplication, is first absorbed in memory and will not prompt a write to the disk until much later. Asynchronous operations, achieved through many layers of caching, invariably blind any deduplication-related signals. Besides such fundamental differences, filesystem deduplication also differs in other important *practical* aspects. For instance, to reduce overhead, the granularity of filesystem deduplication (often as large as 128 KB) vastly exceeds that of memory deduplication (typically 4 KB). For an attacker, large deduplication granularity requires non-trivial massaging when leaking data at a desired byte granularity. Due to these complexities, state-of-the-art storage-based deduplication attacks are limited to exploiting cloud application-level deduplication for cross-user file fingerprinting [25, 40].

In this paper, we present DupeFS, a class of attacks showing that, despite these challenges, exploiting inline filesystem deduplication is feasible. To this end, we describe novel primitives to leak data via filesystem deduplication and analyze their properties. Using these primitives, we build a number of DupeFS attacks, including one leaking arbitrary data at byte granularity. Moreover, we show that we can expand the threat model of such fine-grained attacks from local-only (as done by memory deduplication) to fully remote attacks. This is possible since, in production systems, filesystems are often shared between multiple remote parties, either directly (e.g., a shared file server), or indirectly (e.g., when multiple clients cause a web server to log accesses). In addition, access times to filesystem storage are generally higher than accesses to memory, and, as we will show, even amenable to further amplification by an attacker aware of filesystem internals. This enables remote attacks, where a malicious client leaks secret data from another remote victim client *across the network*.

To craft DUPEFS primitives and obtain secret file data of other users, the attacker generates a carefully-chosen sequence of file operations, specifically tailored to the target filesystem's low-level implementation. The attacks rely solely on timing and storage information available to unprivileged users. We demonstrate the practicality of the side channel with concrete attacks against two popular filesystems, ZFS [9] and Btrfs [47], from different vantage points: local (attacker's code on the victim machine), LAN (attacker across the local-area network), and WAN (attacker across the Internet). For instance, we present an end-to-end DUPEFS attack against an nginx web server running on ZFS/HDD during off-hours. In this scenario, we can leak a (critical) long-lived OAuth access token from the server's access log file over LAN or even WAN at a rate of around 1.5 and 1 byte per hour, respectively.

Finally, we discuss possible mitigations. In particular, we propose to drastically reduce the timing side channel by making filesystem operations that interact with the deduplication subsystem pseudo-constant-time. The goal is to eliminate remote exploitability in a practical way, preserving space savings and avoiding a complete filesystem redesign.

Contributions. We make the following contributions:

- We analyze filesystem deduplication side channels and show that despite the asynchronous disk accesses and large deduplication granularities, attackers can mount byte-level data leak attacks across the network.

- We introduce DUPEFS's novel attack primitives and demonstrate their feasibility in end-to-end attacks to leak data even across the Internet.

- We describe and analyze mitigations for such attacks.

2 Background

A filesystem is the operating system component that controls the storage and retrieval of data to and from storage devices. Compared to DRAM, accessing storage devices is slow. To hide the latency, modern filesystems use a number of optimizations. In particular, *deduplication* finds identical copies of data and stores them as a single shared data *record* of predetermined size. The duplicates are replaced by references to the single record and thus the filesystem reduces the data footprint. More broadly, deduplication is a generic optimization that is used at different levels of the storage hierarchy, including caches [35, 54], cloud services [5, 33], and most importantly, directly in the filesystems themselves [30, 45]. In this paper, we are concerned with the latter.

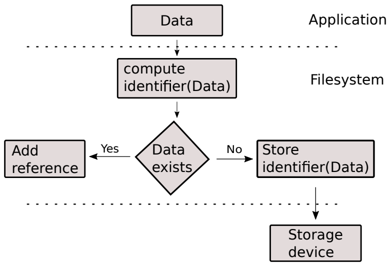


Figure 1: Write path with deduplication.

Basic write workflow. Figure 1 presents a high-level overview of the steps performed by a deduplicating filesystem for an application-issued *write* operation. Upon receiving data from a client application, the filesystem calculates a unique identifier for the data (e.g., by calculating a hash over the content), which it checks against a database of existing identifiers. If none of the existing identifiers match, the new data is written to storage and the identifier added to the deduplication database. If the data exists, the filesystem updates its metadata with a new reference to the existing data and returns control to the application without writing the data to the storage device.

Deduplication mode. In this paper, we are interested in *inline* deduplication, where the filesystem automatically checks for duplicates during the I/O operation—e.g., when data is written. In contrast, offline (or out of band) deduplication is typically a manual process whereby a user runs a deduplication utility explicitly. While inline deduplication introduces some overhead with the identifier lookup, it is the deduplication commonly used in production since, in case of duplication, only a reference is immediately written to storage instead of the duplicate data, leading to space and time savings.

Data identifiers. To identify the content of a deduplication record, the filesystem uses a hash function. Some implementations use collision-resistant cryptographic hash functions such as SHA-256, while others rely on faster hash functions that are not collision-resistant, such as fletcher4 [19]. The same function used for data identification is also used to detect duplicates by computing hashes of candidate data to write and comparing it with the existing deduplication records [16, 31, 45]. Since hashing may incur collisions, some implementations include an additional step to verify that the data inside the matching deduplication records is identical [9].

Deduplication tables. A deduplicating filesystem keeps a history of previously written deduplication records to identify future duplicates. To this end, the filesystem stores the hash values of existing deduplication records as unique identifiers in a data structure called the deduplication table which can be kept in memory, on disk, or both. For inline deduplication, the filesystem accesses the table for every write operation.

Deduplication granularity. Filesystems store a large amount of data. Since the size of the deduplication table is proportional to the total amount of data, filesystems perform deduplication at a granularity (i.e., record size) that is a multiple of the data block size. As a result, a sufficient number of data blocks must be written to the filesystem to reach the deduplication record size before the deduplication checks happen.

3 Threat Model

We assume an attacker who has direct or indirect (possibly remote) access to the same filesystem as a victim, and the filesystem performs inline deduplication. We assume the filesystem to be free of bugs and that the configuration as well as the access control settings are all correct.

We consider different local/remote attack scenarios, with the attacker colocated with the victim on a given machine (*local*), across a local-area network (*LAN*), or across the Internet (*WAN*). The attacker wants to obtain secret data from the victim's files, even though the file permissions prevent direct access. In the local scenario, the attacker interacts with the filesystem through attacker-controlled (unprivileged) programs that write to and read from the underlying storage using low-level system calls such as *write()*, *read()*, *sync()*, *fsync()*. In the remote scenario, the attacker interacts with the filesystem through a program that is not under the attacker control. For example, in the case of a server program, this is possible through valid requests that lead to data being written to storage on the attacker's or victim's behalf. We assume there is no limit to the number of I/O operations that can be performed by running programs or by sending requests to a server program. Remote attacks, unlike local ones, require control over the victim's actions to perform successful attacks, e.g., the attacker forcing a victim web server to write a secret into a log file on the remote filesystem. We will discuss additional attack-specific assumptions in the corresponding sections.

4 Exploiting Filesystem Deduplication

In this section, we discuss two general primitives and the challenges to build attacks over deduplicating filesystems. In Section 7, we will discuss how to craft such primitives for modern file systems such as Btrfs and ZFS.

4.1 Primitives

The timed write primitive. Figure 1 shows that inline deduplication handles the writing of unique data differently from existing data on the write path. The common path consists of computing the data identifier and checking whether it is new. If so, the filesystem inserts both the new identifier and the data itself. In contrast, if the data existed already, it is sufficient to update the metadata with a new reference to existing data, which is considerably cheaper. This timing difference forms the basis for our *timed write primitive*. Similar to the COW timing primitive on memory deduplication [7, 10], this primitive allows attackers to leak whether certain data is present on the filesystem during a write operation. Unlike prior memory deduplication attacks, building filesystem-based timed write primitives is complicated, as we soon discuss.

The timed read primitive. Prior memory deduplication attacks [10, 42] show one can detect whether a memory page is deduplicated via a read-based cache timing attacks. However, this approach is not generally applicable to filesystem deduplication. Deduplicated data from different files end up in distinct physical memory pages as the page cache in popular operating systems such as Linux operates at the file level. To craft a filesystem-based timed read primitive, we observe that if a block of a file becomes deduplicated, its physical location on the disk differs from its surrounding blocks. We use this observation as a basis for our *timed read primitive*. Building it faces certain challenges which we discuss next.

4.2 Challenges

Modern filesystems perform many optimizations to improve performance and reliability, resulting in a number of challenges to craft our timing-based deduplication primitives.

C1. Performance. In filesystems, the I/O operations are mostly asynchronous to hide the latency of the underlying storage and other internal filesystem operations from client applications. For this reason, filesystems cache data which complicates the construction of a timing attack significantly. As we shall see, asynchronous operations may necessitate additional attack preparation steps that massage the cache before measuring time or attempting synchronous I/O.

C2. Reliability. To ensure that the system is in a sane state when it crashes, filesystems typically write metadata along with the user data to ensure the filesystem can be restored to a consistent state when catastrophe strikes. Even if data is deduplicated, the metadata still needs to be written to disk, which interferes with our timing channel. This makes building reliable timed write primitives particularly challenging.

C3. Capacity. To perform deduplication efficiently, filesystems need to maintain an in-memory digest of existing stored data. Given that a large number of digests may introduce unacceptable overhead, modern filesystems perform deduplication only across many blocks that are either temporally or spatially close to each other, clustered together in a deduplication record. This complicates building our primitives in two ways. First, detecting a deduplication event across many blocks is not trivial, especially for the timed write primitive. Second, the large deduplication granularity significantly increases the entropy of any target secret deduplication record.

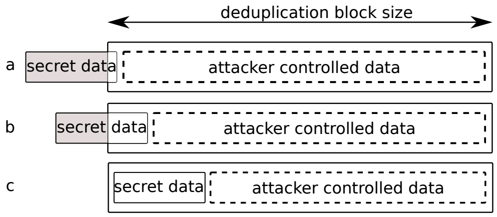


Figure 2: Leaking secret data using deduplication.

5 DUPEFS Overview

To mount DUPEFS attacks, we develop our general primitives into exploitation techniques for three classes of attacks: (i) *data fingerprinting*, (ii) *data exfiltration*, (iii) *data leak*.

5.1 Data fingerprinting

In a data fingerprinting attack, DUPEFS relies on the general timed read/write primitives to reveal the presence of existing known but inaccessible data, such as an inaccessible file of another user. Attackers may use fingerprinting to discover known but embarrassing/compromising content on the server, for instance for extortion purposes. Prior work has shown that timing the writes of a client application may be used for data fingerprinting, but always under the assumption that the client application plays an active role in the deduplication. That is, the client sends data to a cloud application server only if the server does not already have a copy of the data, yielding a timing side channel on write operations [25, 40, 57]. Of course, this is not the case with deduplicating filesystems such as ZFS and BtrFS. DUPEFS shows that similar attacks are still possible, without relying on client applications and application-level deduplication in any way. DUPEFS exploits deduplication performed entirely in the filesystem and is completely agnostic to the applications running on top of the storage stack.

5.2 Data exfiltration

In a data exfiltration attack, DUPEFS relies on the general timed read/write primitives to exfiltrate secret data from a system (or sandbox). The idea is to allow two colluding parties with direct/indirect access to the same system to communicate over a stealthy covert channel. For instance, the parties can use a small number of data blocks with predetermined values to encode messages of a communication protocol [25]. The parties can then exploit timing side channels to find which message was written by the other party. DUPEFS's covert channel can be used to exfiltrate data over LAN or WAN.

5.3 Data leak

In a data leak attack, DUPEFS can leak secret data from a remote system by relying on two exploitation techniques: *alignment probing* and *secret spraying*. The former reduces the entropy of a target secret and enables byte-granular attacks. The latter amplifies the signal and enables remote attacks over LAN/WAN. We first introduce such techniques, then present our example end-to-end data leak attack scenario.

Alignment probing. Figure 2 shows how to exploit alignment probing to leak secret data by carefully aligning known data and then probing for parts of the secret spilled next to it in the same deduplication record (Figure 2-a). By controlling how the data is written to storage, the attacker can stretch controlled data to fill the deduplication record minus one or more bytes of secret data (Figure 2-b). Next, the attacker issues multiple writes with possible guesses for the secret (now low-entropy) record to probe for the unknown byte values until she triggers deduplication (Figure 2-c). At that point, the attacker uses the timed read/write primitive to detect deduplication and hence the correct guess for the unknown byte values. Finally, the attacker repeats the process with multiple alignments until the entire secret is leaked. The attacker relies on the ability to make many instances of the secret appear at various offsets within chunks of otherwise known data. Compared to alignment probing techniques used in prior work in the context of memory deduplication [10], DUPEFS enables such techniques within the filesystem, which is more challenging given the difficulty of enforcing controlled alignment in the storage stack and the coarser block-level interface.

Secret spraying. With basic alignment probing, an attacker can leak part of a secret by timing an I/O operation on a single duplicated or non-deduplicated record. While this may be sufficient for local attacks, remote attacks over LAN/WAN require a stronger signal. To this end, DUPEFS relies on *secret spraying*, a novel deduplication-based exploitation technique for signal amplification. The key idea is to spray candidate secret values over many deduplication records and issue many writes for the corresponding guesses to exploit multiple deduplication events at once. In particular, $N/2$ secret deduplication records and $N/2$ probe deduplication records are carefully crafted with targeted mutations to ensure an attacker can time an I/O operation on $N/2$ deduplicated records (if the guessed probe values are correct) or N non-deduplicated deduplication records (otherwise). Using this technique, DUPEFS can amplify the original number of target deduplication events and thus the signal by a factor of $N/2$.

End-to-end attack. For our example end-to-end remote data leak attack, we target secret data stored in the access log file of a remote (nginx) web server running on top of ZFS/HDD. We specifically target a Single Sign-on (SSO) scenario based on the OAuth protocol [3], where a victim browser accesses an attacker-controlled website with a hidden *iframe* that repeatedly triggers security-sensitive HTTP requests from the victim's browser to an SSO-based service running the nginx web server. In particular, each request URL includes a 22-character OAuth *access token*, which is the target secret stored in the web server's access log file. DUPEFS relies on the primitives and exploitation techniques introduced earlier to repeatedly interact with the web server and leak the token. Section 8 shows how we address all the aforementioned challenges to mount the attack over LAN or WAN. Before that, we provide necessary internal information about ZFS and Btrfs (Section 6), which we use to build the filesystem-specific timed read/write primitives (Section 7).

6 Deduplication in Modern File Systems

In this section, we discuss how modern filesystems such as ZFS and Btrfs perform basic I/O operations, with a focus on deduplication. In practice, deduplication operates at the granularity of multiple disk blocks, a unit that we generally refer to as deduplication record, but that Btrfs calls a *dedupe block* and ZFS calls *record*. To identify deduplication records, these filesystems use a hash function, typically SHA-256, and keep the metadata in hash tables, which, borrowing ZFS terminology, we will refer to as deduplication table (DDT).

ZFS. The Zettabyte File System (ZFS) [9] is a mature transactional copy-on-write filesystem that implements features such as volume management, deduplication, data compression, and snapshots. To support transactions, changes to on-disk data are first inserted in a transaction queue and processed later. Upon transaction completion, ZFS updates the metadata to reflect the changes and finalize the operation. ZFS implements inline deduplication, hence checks for data uniqueness are on the write path, as part of a transaction. ZFS keeps the deduplication table in memory (for ease of access) and on the disk (for reliability). A file in ZFS consists of aligned records of 128 KB in size and deduplication records are also 128 KB by default. It may take multiple transactions to fill a record, but when filled with data, the record becomes deduplicatable—prompting ZFS to look up its hash in the DDT.

Btrfs. The *B-tree* filesystem (Btrfs) [1, 47] is a modern Linux copy-on-write (COW) filesystem that implements features similar to ZFS and uses B-trees along with COW semantics to update the data on the disk. The B-trees are optimized for COW semantics and contain both data and bookkeeping information. The data in Btrfs are stored in *extents*. An extent consists of contiguous, aligned, on-disk data blocks, checksummed for integrity. Like ZFS, Btrfs is transactional. It collects data block changes in memory until the number of collected changes exceeds a threshold or a timeout occurs, at which point it flushes the changes to a new location on the disk. The filesystem state is kept in checkpoints that update the superblock, while extents store metadata such as the file creation checkpoint, the disk area corresponding to a file, the logical offset, and the number of data blocks in the extent. Deduplication in Btrfs works at the level of extents, which become candidate deduplication records when their size reaches the *deduplication record size* of 128 KB (default).

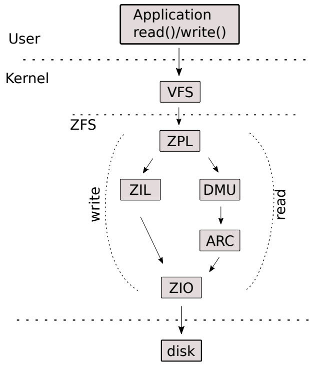


Figure 3: ZFS read/write paths

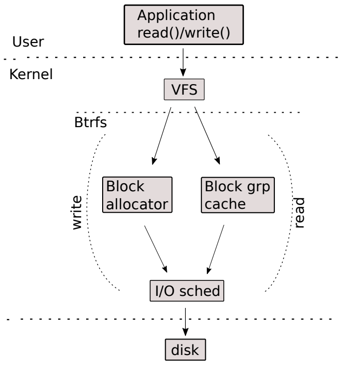


Figure 4: Btrfs read/write paths

6.1 Writes in Deduplicating Filesystems

Write operations in modern filesystems can be either synchronous or asynchronous. An asynchronous write operation does not block the application, which can resume its execution as soon as the data reaches the kernel. Meanwhile, the filesystem dispatches the operation through multiple layers of buffering and writes the data to storage later. In contrast, synchronous writes block and while the data still passes through intermediary buffers, the control returns to the application only after the data is written. When an application calls the *write* syscall, the data is moved from the application's buffers to kernel space, in the page cache of the Virtual File System (VFS), with its pages marked dirty. Next, the kernel dispatches the operation to the specific filsystem (e.g., ZFS or Btrfs).

Writes in ZFS. The left-side of Figure 3 describes the data path for such writes in ZFS. After placing the write in the ZFS Intent Log (ZIL) on disk, the kernel returns from the syscall. The write remains there until, at a later time, ZFS processes the ZIL by passing the data to the ZFS I/O (ZIO) layer and updating the deduplication table. In particular, ZFS uses an on-disk ZIL for reliability and an in-memory ZIL for efficiency. Write requests are committed to the on-disk ZIL in the following cases: the write is synchronous, the application calls the *fsync* syscall, or five seconds have elapsed. Finally, to prevent applications from overwhelming the filesystem, ZFS implements *write throttling*, which temporarily blocks aggressive writers to process outstanding writes.

With respect to the challenges in Section 4.2, challenge C1 stems from the ZIL introducing asynchronous behavior even for synchronous writes as a performance optimization, challenge C2 stems from metadata management in the ZIL and deduplication table for reliability reasons, and challenge C3 stems from the large 128 KB deduplication records that ZFS uses for capacity reasons. Finally, write throttling and ZIL flushing both introduce additional noise.

Writes in Btrfs. Figure 4 describes the equivalent data paths in Btrfs, with the write operation passing through the deduplication checks when the filesystem writes the data to disk [13].

In the default filesystem settings [14], Btrfs deduplicates data using the inline *deduplication record size* of 128 KB. After looking up the identifier in the deduplication table, it writes both data and metadata to disk if the data is new, or only the metadata if the data already exists.

The extents in Btrfs are contiguous on-disk data blocks and each file consists of one or more extents. In case of a large write, only the full extents are candidates for deduplication, while any remaining bytes become an extent with a smaller size. If the application subsequently appends data to the file, the new data is not merged with the small extent, but placed in a new extent. Btrfs does not support modifying or splitting extents and a write in an existing extent will trigger the creation of a new extent with the new data and an update of the file indexing information. For instance, when an application overwrites the first 100 bytes of a file, the copy-on-write behavior creates a new extent of one disk block which contains the new 100 bytes plus the rest of the first disk block of the original extent. When the file is read, Btrfs returns the first disk block of the new extent while taking the remainder of the data from the original extent.

With respect to the challenges identified in Section 4.2, we see again that the asynchronous transaction introduces challenge C1, the metadata handling for reliability introduces challenge C2, and the large deduplication records introduce challenge C3. Furthermore, partially filled extents, if any, and alignment issues complicate the attack.

6.2 Reads in Deduplicating Filesystems

When the kernel handles a *read* syscall, it retrieves the data either from the filesystem cache or from the disk.

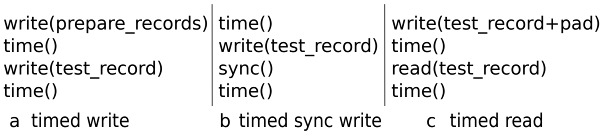
Reads in ZFS. After the read syscall has passed through the VFS layer and the ZFS Posix Layer, ZFS checks if the data exists in the *Adaptive Replacement Cache*(ARC) and, if so (right-hand side of Fig. 3), returns the data to the application. Otherwise, it transfers control to the ZFS I/O (ZIO) layer which retrieves the data from the disk.

Reads in Btrfs. In Btrfs, after the read syscall has passed through the VFS layer, Btrfs performs a search in the "block group cache". In case of a miss, Btrfs retrieves the data from the disk. As mentioned, Btrfs may create new, partially-filled extents in case of file modifications and, in that case, the read may access more extents than one would expect.

For both filesystems the COW behavior creates an on-disk layout that is non-sequential for deduplicated data and typically sequential otherwise. Reading the contents of a file with deduplicated data involves random accesses on the disk which is usually measurably slower than sequential accesses. However the partially filled extents that result from file modifications also incur non-sequential accesses and generate noise.

7 Attack Primitives

This section introduces the general timed write/read primitives for ZFS and Btrfs to perform DUPEFS attacks. To exploit the timing side channel, an attacker writes data to the filesystem using carefully crafted patterns that massage the filesystem into an exploitation-friendly state. In particular, to handle *transactional behavior* potentially delaying write operations, the attacker performs multiple writes to flush each transaction. To handle *copy-on-write behavior* and the *coarse deduplication granularity*, the attacker writes a sufficient amount of data to trigger deduplication checks. To handle *caching behavior*, the attacker uses *sync* operations to force a cache flush where possible (locally on Btrfs) or massages the cache with enough I/O operations otherwise. Finally, to handle *concurrent operations* from other applications, the attacker repeatedly measures filesystem operations to confirm deduplication.



```
write(prepare_records)
time()
write(test_record)
time()
```

a timed write

```
time()
write(test_record)
sync()
time()
```

b timed sync write

```
write(test_record+pad)
time()
read(test_record)
time()
```

c timed read

Figure 5: DUPEFS attack primitives

Overview. Figure 5 details our attack primitives in pseudocode. Each primitive describes I/O operations on multiple deduplication records to trigger deduplication checks. The test deduplication record is the record of 128 KB (by default) an attacker writes to the filesystem and then measures the impact of the operation in the time domain (by timing the write itself or subsequent operations). In the write operation, the test deduplication record can be padded with other deduplication records that help with filesystem massaging (e.g., flushing caches), alignment probing, and secret spraying. Depending on the attack, the content of the test deduplication record is known (e.g., data fingerprinting) or mostly known except for, say, 1 byte the attacker needs to guess (e.g., data leak).

In details, Figure 5-a presents a timed write primitive (available on ZFS). First, the attacker writes multiple controlled deduplication records to the filesystem in a *prepare phase*. This is to cause a transaction cache flush (absent *sync* support in ZFS) and prepare alignment. Next, the attacker issues (and times) a write of the test deduplication record.

Figure 5-b presents a timed synchronous write primitive (available on Btrfs). First, the attacker issues a *sync* operation to flush the caches. Next, the attacker issues (and times) a write of the test deduplication record. This primitive is available only in exploitation scenarios where the attacker can trigger *sync* operations. This is nontrivial in remote data leak attacks, where the only (unlikely) option is to lure a victim server program into issuing an explicit *sync* operation.

Figure 5-c presents a timed read primitive (available on Btrfs), which can probe for deduplication events after the fact. First, the attacker writes a test deduplication record (plus padding) to the filesystem, which may trigger deduplication. Next, the attacker reads back the same data. If deduplication did happen, the (random-access) read will take longer than the (sequential-access) read for the nondeduplicated case.

In the next section, we analyze the timing side channel for the different primitives on an SSD (Corsair Force LS SSD S9FM02.6) and a magnetic HDD (model ST1000). The latter is obviously slower, but still a popular type of storage medium, especially in server environments [6]. Unless otherwise noted, we consider default configurations of ZFS and Btrfs, with a deduplication record size of 128 KB. We repeat experiments several times and find marginal deviations in results.

Timing differences on Btrfs. To verify the existence of the timing side channel on Btrfs, we evaluate Btrfs running on Linux (v4.20). In our experiments, we issue 500 *synchronous* write operations (thanks to the Btrfs-supported *sync*) of identical (deduplicated) deduplication records and 500 write operations of unique deduplication records. We obtain an average timing difference between deduplicated and unique write operations of 0.57 ms for the SSD and 24.5 ms for the HDD. Repeating the same experiment with asynchronous write operations leads to no statistically meaningful difference. As such, we do not further consider this configuration in our analysis. Nonetheless, this experiment confirms a realistic signal for our (synchronous) Btrfs write primitive on Linux.

On the same setup, we issue 500 read operations for deduplicated records and 500 read operations for nondeduplicated records. We obtain an average timing difference between deduplicated and nondeduplicated read operations of 0.7 ms for the SSD and 17.22 ms for the HDD. This experiment confirms a realistic signal for our Btrfs read primitive on Linux.

Timing differences on ZFS. To verify the existence of the timing side channel on ZFS, we first consider ZFS running in its natural FreeBSD (10.4) habitat. In our experiments, we perform asynchronous write operations using both identical (i.e., deduplicatable) and unique records. We write enough data in the 5 second time interval before the in-memory ZIL is flushed to disk and measure the time to complete the individual write operations.

We issue 500 write operations of identical (deduplicated) records and 500 write operations of unique records. We obtain an average timing difference between deduplicated and unique write operations of 0.04 ms for the SSD and 2.6 ms for the HDD. This experiment confirms a realistic signal for our ZFS write primitive on FreeBSD. While the SSD signal seems weak at first glance, this is just due to the larger default transaction cache size on ZFS. This simply means we need a larger number of writes for a strong signal. We confirm this by repeating the SSD experiment with a smaller transaction cache of 10 deduplication records and measuring a timing difference of 1.23 ms.

Due to different licensing models (CDDL vs. GPL), ZFS is not directly included in the Linux kernel. As a result, the Linux implementation contains more software layers that collectively dampen the signal for our timing side channel. To confirm this intuition, we re-run our last experiment on Linux (v4.20)-based ZFS and report an average timing difference between deduplicated and unique writes of 0.16 ms on HDD and no statistically meaningful difference on SSD. Similarly, ZFS' efficient read implementation does not yield a meaningful signal on HDD or SSD. As such, we do not further consider Linux/ZFS or ZFS-based read primitives in our analysis.

8 DupeFS Exploitation

To illustrate the severity of DupeFS, we exemplify attacks for *data fingerprinting*, *data exfiltration*, and *data leakage*.

8.1 Data fingerprinting

We exemplify a data fingerprinting attack using our Btrfs-based synchronous write primitive in a local exploitation scenario[1]. A local unprivileged attacker seeks to detect the existence of an inaccessible file (or deduplication record within a file) with known content. This is useful to detect specific system binaries or configuration files and fingerprint vulnerable programs running on the victim system. The attacker first prepares an oracle of target files with a size matching or larger than the deduplication record size. Next, the attacker runs an unprivileged program on the target system to repeatedly effect the timed write primitive for each file. Using the *syncfs* and *write* syscalls, the attacker synchronously writes each file to the victim filesystem and times the operation to detect deduplication indicating the presence of the file.

8.2 Data exfiltration / covert channel

We exemplify a data exfiltration attack using our Btrfs-based synchronous write primitive in a local exploitation scenario[2]. A local unprivileged attacker (or "sender") seeks to exfiltrate data from a sandbox over a covert channel. The receiver is an unprivileged colluding party running on the same system. For simplicity, the two communicating parties run a basic covert channel protocol and synchronize using the system clock.

First, the sender writes N deduplication records for each bit of data to a file. Each record is filled with a predetermined deduplication record prefix, a $[0...N-1]$ deduplication record ID, and the $[0-1]$ bit value. Next, the receiver uses the timed write primitive on another file in order to test for each unknown bit value across the same number of (N) deduplication records. The receiver uses the same record format as the sender and tests for 0-bit deduplication records. A signal (or its absence) determines the transfer of a 0-bit (1-bit) value.

[1] We have also reproduced the attack on the ZFS write primitive

[2] We have also reproduced the attack on the ZFS write primitive

After the receiver has stored the leaked bits, the protocol repeats with the sender writing new data-encoding deduplication records. For example, to exchange 1 byte of information using $N = 10$, the sender writes $10 * 8$ deduplication records to a file. The receiver then uses the timed write primitive with $10 * 8$ 0-bit test deduplication records on a separate file. Fast (deduplicating) writes on the first 10 deduplication records signal a 0 value for the first bit, slow (nondeduplicating) writes signal a 1 value for the second bit, and so on. We use multiple (N) records to transfer a single bit to amplify the signal and thus reduce the error rate of the covert channel. We use different deduplication record IDs to prevent the deduplication records written by the sender (or receiver) from deduplicating against themselves.

8.3 Remote data leak

We exemplify a data leak attack using our ZFS-based write primitive in a remote exploitation scenario[3]. The attacker seeks to leak an OAuth access token [3] from the access log of a remote nginx web server running on ZFS/HDD [4]. The nginx web server hosts a website (e.g., https://someapp.com) which the victim has granted access to an authenticated third-party service (e.g., https://github.com) via long-lived OAuth access tokens. At a high level, to implement the attack, the attacker lures a victim browser into an attacker-controlled website, which repeatedly (but transparently to the user) forces the browser to access the nginx web server with the secret OAuth access token encoded in the URL. As such, the secret is repeatedly spilled on the nginx access log stored on ZFS, enabling alignment probing and secret spraying. Meanwhile, the attacker concurrently and independently probes the nginx web server to leak the secret OAuth token one byte at the time. With the token, the attacker can gain access to the third-party service with the victim's credentials.

Attack scenario. The end-to-end attack scenario in our example has the following actors: the victim browser, a Single Sign On (SSO) server, an SSO client running nginx on top of ZFS, a third-party service the SSO client has been granted access to on the victim's behalf via SSO access tokens, and a malicious website under the control of the attacker (say https://attacker.com). Specifically, we target SSO access tokens from the OAuth 2.0 implicit grant access scheme [3] and assume such access tokens do not expire for the entire duration of the attack. This is a sensible assumption, as many third-party services use long-lived access tokens that never expire [2]. To mount the attack, the attacker-controlled website includes a hidden *iframe* which repeatedly forces the browser to connect to the SSO client with the secret OAuth token. This is legal behavior, but defenses against clickjacking, *X-Frame-Options* (XFO) in particular, may prevent such accesses from an *iframe*. In our example attack scenario, we assume the victim server does not offer XFO (which is very common even in modern websites [26]), or the victim browser's XFO implementation is bypassable (e.g., Microsoft Edge [15]).

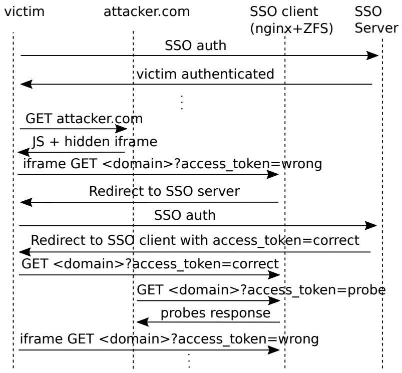

victim
attacker.com
SSO client (nginx+ZFS)
SSO Server
SSO auth
victim authenticated
GET attacker.com
JS + hidden iframe
iframe GET <domain>?access_token=wrong
Redirect to SSO server
SSO auth
Redirect to SSO client with access_token=correct
GET <domain>?access_token=correct
GET <domain>?access_token=probe
probes response
iframe GET <domain>?access_token=wrong

Figure 6: Data leak attack request sequence

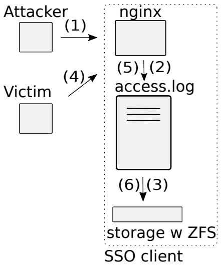


Figure 7: Data flows in the SSO client

Attack workflow. Figure 6 presents all the requests exchanged between the victim browser, the attacker-controlled website, the SSO client, and the SSO server before and during the attack. Before the attack, the victim has already obtained an OAuth access token for the third-party service by authenticating with the SSO server. In the first stage of the attack, the victim browser is lured to the attacker-controlled website. The website serves the victim browser some attacker-controlled JavaScript and a hidden iframe. The latter issues an HTTP request to the SSO client using an incorrect access token.

Upon receiving the incorrect access token, the SSO client redirects the iframe of the victim browser to the SSO server to obtain a new valid access token. When the SSO server receives the request, it simply acknowledges that the victim browser is already authenticated and redirects the iframe again to the original SSO client. The redirect causes the iframe to issue an HTTP GET request with the correct (and secret) OAuth access token to the SSO client. Since the access token is encoded in the URL and the URL is logged in the access log of the SSO client's nginx web server by default, the request ultimately spills the target secret on the victim ZFS filesystem. At that point, the attacker can independently issue multiple GET requests to the SSO client in order to probe for the secret byte values of the access token. Meanwhile, the malicious JavaScript from the victim browser can reload the iframe and the entire sequence repeats, until the attacker obtains the secret access token using the timed write primitive for ZFS across the network.

Inside the SSO client. We now focus on the SSO client, which runs the nginx web server and stores its access log on the target ZFS filesystem. The requests that the attacker directly or indirectly sends to the SSO client reach nginx, which, in turn, uses system calls to write each HTTP request to the access log. The data flows for the attacker and victim inside the SSO client are shown as (1)-(3) and (4)-(6) in

[3] This primitive provided the best signal for remote attacks

[4] In our experiments, the HDD setup was necessary for a realistic signal across the Internet

Figure 7. The sequence of events above results in different classes of attacker-controlled entries being spilled into the access log: initial GET requests issued by the browser with an incorrect access token, second-stage GET requests issued by the browser with the correct access token, GET requests issued and timed by the attacker to probe for the secret bytes.

The attacker carefully massages the workflow above to interleave the different classes of GET requests and implement the required primitives. In particular, the attacker first issues a number of wrong-access-token requests causing nginx to carefully align the access log entries. By massaging the alignment (starting from a baseline of known access log alignment leaked with timed write primitive), the attacker can ensure the next GET request with the correct access token fills an entire access log deduplication record and spills the last byte of the access token into the next deduplication record. The attacker then performs additional GET requests to nginx to fill the rest of the deduplication record. At that point, the attacker can time specially-crafted GET requests to probe for each of the possible byte values. Figure 9 presents the access log file layout of nginx induced by the proposed attack patterns. Since OAuth uses 22-character access tokens, the entire process is repeated 22 times [3], leaking 1 byte value (from the base64 alphabet) of the access token per iteration.

Since we control both the browser- and the attacker-issued requests to nginx (using controlled iframe refreshes or independent client requests from an attacker-controlled machine), we can repeat the individual steps many times. This enables secret spraying for amplification (simply mutating the original wrong-access-token requests to cause the browser to send different variations of the correct-access-token requests to nginx) and repeated alignment probing (shifting the alignment by 1 byte every time) to leak all the bytes of the secret.

For every byte value of the secret access token probed, the attacker runs a number of P probes per byte using the pattern described in Figure 9 and measures the time to complete the individual network requests. Whenever the in-memory ZIL is flushed to disk a peak can be observed in the timing measurements of the attacker as feedback directly from the VFS layer. The attacker can analyze the real-time duration of the ZIL flush by observing the peaks in real time. The duration of the peak is larger when nondeduplicated data is flushed than when there is deduplicated data. To amplify the difference the attacker uses the secret spraying technique to submit large amounts of data to ZIL. Figure 8 shows an example of the width (duration) difference for 2 consecutive peaks, for both the deduplicated and nondeduplicted case, separated by a 5-second interval. The attacker picks the byte value that produces the peak with a duration that is below a threshold as the correct byte value of the secret access token. The threshold is discovered empirically by the attacker (e.g., a peak duration of at least 0.5s to signal the correct byte value for the data in Fig. 8) and depends on the network bandwidth.

To improve performance of the logging feature, nginx uses an intermediary buffer which collects I/O operations before sending them to ZFS. The buffer is limited to 5 requests irrespective of their size and works as a FIFO queue: once a new request comes in, the oldest request is pushed out and submitted to the VFS layer to be written to disk. To deal with buffering, the attacker floods the intermediary buffer and controls when the write operation (including deduplication checks) is performed. In fact, nginx' use of the intermediary buffer helps the attacker, as it is easier to control alignment (within the buffer) and generate single large writes.

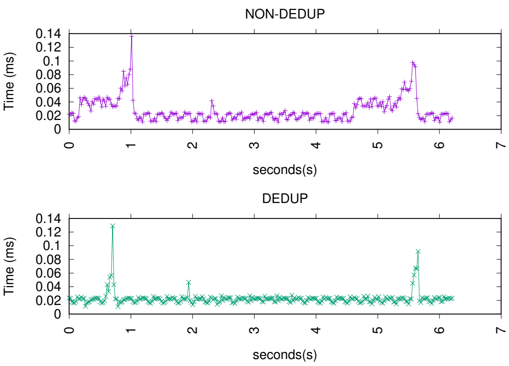


Figure 8: Dedup vs non-dedup peaks: the duration differs noticeably

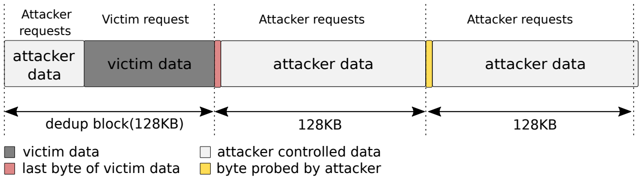


Figure 9: Crafted attack patterns in the nginx access log

Note that the attacker controls most of the data that is written to the log, namely: the url of the requested resource, the user agent field, the request body, etc., which allows the attacker to reduce the entropy of probing to that of the secret. Some data, such as the timestamp, is not directly controlled by the attacker. However, the attacker learns it, at second granularity, from the Date header field in the HTTP reply—which is mandatory according to RFC2616. Based on the acquired timestamps, the attacker can synchronize the requests with the server, eliminating entropy.

9 Evaluation

We evaluate our DUPEFS attacks on a system equipped with an Intel Core i5-8250U CPU (4 CPU cores), 16 GB of RAM, an SSD (Corsair Force LS SSD S9FM02.6), and a magnetic HDD (Seagate ST1000). We run our target filesystem implementations in their default settings (with a deduplication record size of 128 KB) and on their natural operating system platforms, namely FreeBSD (10.4) for ZFS and Linux (v4.20) for Btrfs. ZFS, in particular, uses the default amount

Table 1: File fingerprinting

File	Type	Size	Success
config-4.11.3-200.fc25.x86_64	text	181 KB	70%
lena_color.gif	binary	223 KB	55%
libz3.so	binary	22 MB	99%
x86_64-redhat-linux-c++	binary	1 MB	99%

Table 2: Covert channel

N	Bit errors	Time	BR	BER	I/O
20	13	375s	0.320 bps	10.83%	76.8MB
40	14	746s	0.160 bps	11.66%	153.6MB
60	12	1591s	0.075 bps	10.00%	230.4MB
100	6	1873s	0.064 bps	5.00%	384.0MB
120	3	2387s	0.050 bps	2.50%	460.8MB

of memory for dirty data (10% of the RAM size) and its reliability configuration using sync=*always*, which ensures that the in-memory ZIL is also saved to disk—enabling recovery after a crash. We repeat all experiments multiple times on a quiescent server and report the mean values.

9.1 Data Fingerprinting

We evaluate our data fingerprinting attack using the Btrfs-based write primitive over HDD. We have reproduced these experiments on SSDs and using the ZFS write primitive, observing similar results, which we omit for space reasons. For our experiments, we consider an unprivileged attacker interested in probing for a number of sensitive files (larger than the deduplication record size of 128KB) on the running sytem.

Table 1 presents our results. We consider 3 different binary files (a picture, a shared library, and a binary executable) and one text file (the kernel configuration file) for our analysis. Note that, most of the contents of the kernel configuration file is predictable, given that each line normally refers to a configuration OPTION in one of the following 4 variants:

```
1.    #CONFIG_OPTION is not set
2.    CONFIG_OPTION=y
3.    CONFIG_OPTION=m
4.    CONFIG_OPTION=n
```

DUPEFS reliably figerprints individual (128 KB) fragments of the target files. The table presents success rates for fingerprinting the entire file. DUPEFS can reliably fingerprint the target data except the last sub-128 KB chunk of a file. Thus the small (181 KB and 223 KB) files have lower success rates.

9.2 Data Exfiltration

We now evaluate our data exfiltration attack, again using the Btrfs-based write primitive over HDD. As we shall see, other configurations again yield comparable results. For our experiments, we consider two unprivileged colluding parties running on the same machine. Both parties exchange information using the covert channel protocol introduced earlier.

Table 3: LAN 1 byte data leak

Success	Attack time/byte	Probes/byte val	I/O
50%	19.2 min	200	4.9 GB
80%	25.6 min	300	7.3 GB
92%	42.6 min	400	9.8 GB
96%	78.9 min	800	19.6 GB

Table 4: WAN 1 byte data leak

Success	Attack time/byte	Probes/byte val	I/O
64%	24.5 min	200	4.9 GB
87%	38.4 min	300	7.3 GB
94%	59.7 min	400	9.8 GB
94%	110.9 min	800	19.6 GB

To evaluate the channel, we transfer chunks of 15 bytes from the sender to the receiver, measuring the bit rate and bit error rate. We repeat the measurements for different numbers of deduplication records (N, determining the number of probes per bit) to investigate the throughput/reliability tradeoff.

Table 2 presents our findings, including the amount of I/O involved in the transfer. Our results show that the bit rate starts from 0.32 bit/s for 20 probes per bit with a bit error rate of 10.83% and drops to 0.05 bit/s for 120 probes per bit with a bit error rate of 2.5%. We also reproduced these results on SSDs and using the ZFS write primitive, with a proportional signal, matching the trend detailed in Section 7. Our results confirm the covert channel can be used for realistic data exfiltration attacks. Note that the bit errors in the covert channel can be compensated by running a simple error correction protocol.

9.3 Data leak

We now evaluate our remote data leak attack, using the ZFS-based write primitive. In many environments, this would be the most worrying attack. The goal of the attacker is to leak the access token, as used commonly on the web, from an SSO client *across the network*. The SSO client runs nginx version 1.14.0_12,2 with the default settings, logging HTTP requests to the access log, over ZFS/HDD. We consider 2 locations for the attacker: one where the attacker is on a wide area network (WAN), far from the server, and one where the attacker is in the same local network (LAN). In the WAN attack, the attacker is located 12 hops away from the victim with an RTT of 2 ms, measured using traceroute with TCP SYN probes. In the LAN scenario, the attacker is located 1 hop away from the victim with an RTT of 0.1ms. The attacker probes for OAuth secrets of 22 bytes encoded in base64.

To evaluate the attack success rate and attack time, we vary the number of probes per byte value from 200 to 800. Tables 3 and 4 present the success rate (out of 50 attempts) and the total attack time for 1 byte in a LAN and WAN setting.

LAN attack. Table 3 presents the success rate to discover 1 byte over a LAN, the time needed to leak 1 byte given the number of probes/byte value used, and the amount of I/O (in GB) used in this attack scenario. The attacker can tune the attack to obtain a desired attack performance-reliability tradeoff. Given a success rate of 92%, the attacker, using the configuration of 400 probes/byte value, leaks 1 byte over the network in roughly 42 min and the full 22-character OAuth access token in around 15 hours.While high success rates require substantial amounts of I/O, such attacks are already within reach of attackers today and will be even more so as the speed of file systems and networks increases.

WAN attack. Table 4 presents the success rate and time needed to guess 1 byte over a WAN, given the number of probes per byte value used, and the corresponding amount of I/O. As shown, the attacker has different options to select the reliability-performance tradeoff for a desired success rate. For example, for a success rate of 94%, the attacker can use 400 probes/byte value, resulting in approx 1h to leak 1 byte and around 21h to leak the full 22-character OAuth access token.

Noise. As well-established in literature [18, 20, 21, 42], the signal progressively degrades in the presence of noise (i.e., concurrent I/O workloads). As a result, in noisy environments, the attack, when not conducted during off-peak/idle times, would require more repetitions and hence more time [20]. For example, in a LAN setting, we generated concurrent load by continuosly reading data from `/dev/random` and writing it to disk. The attack used 800 probes per byte value. The attack duration was ≈91 min (up from 78.9 min), the success rate dropped to 90% (down from 96%), and the I/O performed was 19.6 GB by the attacker and 6 GB by the script.

10 Mitigation

Similar to prior side-channel attacks, DUPEFS attacks are not very stealthy and could be detected by an intrusion detection system (IDS) monitoring I/O activity. Nonetheless, as observed in literature [50], it is difficult to design such an IDS to guarantee no false negatives and no false positives in practice. As such, we now consider more principled mitigations that can provide security-by-design guarantees.

An ideal implementation of filesystem deduplication would save space and have constant-time behavior. In other words, all the deduplication-aware I/O operations need to implement a *same-behavior* policy [42]. This essentially translates to each operation traversing the storage stack in the same amount of time regardless of whether data handled by the operation has been deduplicated or not. In practice, a strict same-behavior policy is neither desirable—as it would hurt space savings—nor practical—as it would not only require a redesign of the filesystem, but also of the physical storage devices. Our goal here is instead to discuss a practical, *pseudo-same-behavior*, mitigation strategy that drastically reduces the (I/O-based) signal and deters remote attacks.

A mitigation for the write path would change the behavior described in Figure 1 for the case when the deduplication checks conclude that the data exist to update the reference and then still perform the write operation to the disk. The duplicate data is simply overwritten. To investigate the practicality of this strategy, we have experimented with Btrfs implemented in the Linux kernel (v4.20) [12].

Write path. In Btrfs, the *submit_compressed_extents* function contains the program point where the write code path diverges in a deduplication-dependent way and induces different behavior in the time domain. Inside this function the block allocation is followed by the *dedupe_hash_hit* check which determines whether to finalize deduplication or else write a nondeduplicated block to disk. To bring the implementation as close to the same-behavior policy as possible with few code changes, we propose a patch to also perform the write operation on the else branch of *dedupe_hash_hit*, simply overwriting existing on-disk data. With a 5 LOC change, we preserve space savings, only slow down deduplicated write paths (mirroring the execution time of non-deduplicated write paths), and eliminate the classic deduplicated write path side channel. We have verified the proposed strategy is sufficient to cripple the SSD/HDD signal for remote attacks.

To verify the performance impact of our proposed mitigation we ran microbenchmarks on a system with an SSD, using 5,000 synchronous write operations with deduplicated data (worst-case scenario)—with and without our mitigation enabled. When the mitigation is enabled, the median performance overhead is as low as 6.7% compared to the mitigation disabled case. Note that performing redundant I/O reduces device longevity compared to the `deduplication=on` baseline (but is equivalent to the `deduplication=off` baseline).

Read path. For this path, the mitigation has to enforce pseudo-same-behavior for disk access patterns. For this purpose, we need to patch the *btrfs_readpages* function, which reads the extents of a file. Since a strict same-behavior policy would require random access for each read operation (with possible performance loss), our strategy here is to introduce time jitter on the read path. We implemented this strategy with a 2 LOC change. We have verified (by running similar microbenchmarks as done for the write path) that even low jitter values are sufficient to cripple the SSD/HDD signal for remote attacks, while introducing no observable performance impact. Note that applying the same jitter-based mitigation on the write path is, in contrast, ineffective (as we have experimentally confirmed), since the write-path signal is too strong to be efficiently eliminated using jitter.

Limitations. With less than 10 LOC changed in the large Btrfs codebase, we believe our mitigation proposal is practical and has a chance at mainline inclusion. Nevertheless, we emphasize these changes only seek to deter remote attacks but cannot completely eradicate the signal for local attacks. For instance, the write path mitigation only enforces a same-behavior policy from the disk perspective. It does *not* eradicate all the code differences on the write path. We believe this limitation still offers a good compromise, since, on a local setting, there are already more powerful side channels (e.g., cache side channels) to mount practical end-to-end attacks.

Another limitation is the mitigation operating only in the time domain (similar to prior secure memory deduplication systems [42]). There may be other side channels that escape our same-behavior policy in other domains. For instance, tools reporting free disk space information to unprivileged users may re-enable very reliable (local) attacks. Free disk space or similar leaky filesystem information should be restricted to privileged users to deter practical side-channel attacks.

11 Related Work

Deduplication is used to efficiently store data in different types of memory, ranging from desktop computers [4, 39] to caches [35, 54] and cloud services [5, 33].

Deduplication Attacks. Many recent efforts investigate the security of deduplication from both an offensive and defensive perspective [27, 28, 34, 40, 44, 46]. Existing storage-based attacks exploit deduplication in the cloud application layer, mostly to detect the presence of particular files. Harnik et al. [25] describe deduplication-based attacks to identify files on the cloud side by observing the amount of data transferred by the client. Mulazzani et al. [40] exploit the hashing mechanism of the Dropbox storage provider to obtain information about the existence of a file. Access to it can be obtained by providing the hash to the service. The attacks exploit the application-level cross-user deduplication performed by Dropbox. In contrast, DUPEFS targets low-level deduplication in modern filesystems, enabling application- and cloud-agnostic attacks leaking arbitrary byte-granular data.

Memory deduplication is a technique used by modern hypervisors or operating systems to reduce main memory usage. Memory deduplication attacks can locally fingerprint applications [22, 52], operating systems [43], or defeat ASLR [7]. Dedup Est Machina is a more advanced memory deduplication attack [10], which can read arbitrary data from the local system's memory using alignment probing and other memory-specific exploitation techniques. In contrast, DUPEFS repurposes alignment probing to exploit filesystem deduplication and combines it with secret spraying to enable byte-granular data leak attacks across the network for the first time.

Deduplication Defenses. Rabotka [46] identifies 4 classes of countermeasures against traditional storage-based deduplication attacks: encryption enforced by the client [37, 51] or by a third party [56], noise added by probabilistic uploads [24, 25, 60], proof of ownership [24, 58], and obfuscation enforced by an intermediate gateway in the network [27]. All these mitigations are only applicable to cloud application scenarios where the client plays an active role in the deduplication. As such, they are ineffective against DUPEFS' attacks based on filesystem deduplication. VUSion [42] proposes a memory deduplication redesign based on same-behavior (i.e., constant-time sensitive operations) and other principles to cripple both side-channel and Rowhammer attacks. In contrast, we show enforcing a pseudo-same-behavior policy—sufficient to deter remote attacks—is feasible with small changes rather than a complete filesystem redesign.

Network Side-channel Attacks. Many prior efforts propose remote network side-channel attacks. Early attacks leak sensitive (cryptographic) data but only target vulnerable server applications [8, 11, 41]. More recently, NetSpectre [48] and NetCAT [32] exploit cache side channels over the network. The former targets a vulnerable (or cooperative) server application containing specific gadgets, while the latter assumes specialized hardware (Intel DDIO and RDMA)—similar to state-of-the-art network-based Rowhammer attacks [36, 53]. Page cache attacks [23] can exploit the operating system's page cache to implement a covert channel between cooperating parties over the network. In contrast to all these attacks, DUPEFS can target arbitrary noncooperative applications running on top of a commodity hardware/software stack and can leak sensitive byte-granular data over LAN/WAN. In concurrent work, Schwarzl et al. [49] showcase similar remote attacks exploiting memory (rather than) storage deduplication and operate byte-by-byte disclosure at comparable speeds.

12 Conclusion

In this paper, we showed that deduplication in commodity filesystem implementations poses a nontrivial security threat. Specifically, we presented evidence that such implementations yield timing side channels that can be abused to remotely leak arbitrary data at byte granularity. To substantiate our claims, we presented DUPEFS, a class of filesystem deduplication-based attacks for remote data fingerprinting, exfiltration, and disclosure. Our end-to-end data leak attack demonstrates DUPEFS can disclose sensitive data from a remote server program even across the Internet. Finally, we investigated mitigations and showed that implementing a pseudo-same-behavior policy for all the I/O operations in the time domain is practical without a full filesystem redesign.

Disclosure

We have disclosed our findings to the affected parties.

Acknowledgements

We thank our shepherd, Carl Waldspurger, and the anonymous reviewers for their comments, as well as Ilias Diamantakos for early signal testing. This work was supported by the EU's Horizon 2020 programme under grant agreement No. 825377 (UNICORE), Intel Corporation through the Side Channel Vulnerability ISRA, and NWO through project "Intersect".

References

[1] btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[2] Github: Creating a personal access token for the command line. https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line.

[3] The oauth 2.0 authorization framework. https://tools.ietf.org/html/rfc6749.

[4] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Trans. Storage*, 2007.

[5] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. Transparent Data Deduplication in the Cloud. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 886–900, New York, NY, USA, 2015. ACM.

[6] Backblaze. Backblaze hard drive stats q2 2019. https://www.backblaze.com/blog/hard-drive-stats-q2-2019, 2019.

[7] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT'15, 2015.

[8] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, 2005.

[9] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. https://pdfs.semanticscholar.org/27f8/1148ecbcd04dd97cebd717c8921e5f2a4373.pdf, 2003.

[10] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16, 2016.

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[12] Btrfs Contributors. Linux fork for in-line dedupe. https://github.com/littleroad/linux/tree/dedupe_latest, 2019.

[13] btrfs Wiki. Dedupe design notes. https://btrfs.wiki.kernel.org/index.php/Design_notes_on_dedupe.

[14] btrfs Wiki. User notes on dedupe. https://btrfs.wiki.kernel.org/index.php/User_notes_on_dedupe.

[15] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent click-jacking protection on the web. In *USENIX Security*, 2020.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[17] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, Santa Clara, CA, February 2016. USENIX Association.

[18] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. CCS'14, 2014.

[19] John Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247 – 252, January 1982.

[20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[21] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. NDSS'17.

[22] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. ESORICS'15. 2015.

[23] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *CCS*, 2019.

[24] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[25] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security Privacy*, 8(6):40–47, November 2010.

[26] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. Tricking johnny into granting web permissions. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 276–281. 2020.

[27] O. Heen, C. Neumann, L. Montalvo, and S. Defrance. Improving the resistance to side-channel attacks on cloud storage services. In *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*, 2012.

[28] J. Hur, D. Koo, Y. Shin, and K. Kang. Secure data deduplication with dynamic ownership management in cloud storage. *IEEE Transactions on Knowledge and Data Engineering*, 2016.

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Jackpot stealing information from large caches via huge pages. Cryptology ePrint Archive, Report 2014/970, 2014. https://eprint.iacr.org/2014/970.

[30] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[31] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[32] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *S&P*, 2020.

[33] W. Leesakul, P. Townend, and J. Xu. Dynamic Data Deduplication in Cloud Storage. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 320–325, 2014.

[34] Jin Li, Xiaofeng Chen, Fatos Xhafa, and Leonard Barolli. Secure deduplication storage systems supporting keyword search. *J. Comput. Syst. Sci.*, 2015.

[35] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, 2016. USENIX Association.

[36] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[37] Jian Liu, N. Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[38] Dirk Meister, Andre Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 175–182, San Jose, CA, 2013. USENIX.

[39] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 2012.

[40] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark Clouds on the Horizon: Using Cloud Storage As Attack Vector and Online Slack Space. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[41] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. Cache time-behavior analysis on aes. *Selected Area of Cryptology*, 2006.

[42] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure Page Fusion with VUsion. SOSP'17.

[43] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. IPCCC'11, 2011.

[44] P. Puzio, R. Molva, M. Önen, and S. Loureiro. Cloudedup: Secure deduplication with encrypted data for cloud storage. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013.

[45] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[46] Vladimir Rabotka and Mohammad Mannan. An evaluation of recent secure deduplication proposals. *J. Inf. Secur. Appl.*, 2016.

[47] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[48] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[49] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *NDSS*, 2022.

[50] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *CCS*, 2004.

[51] Z. Sheng, Z. Ma, L. Gu, and A. Li. A privacy-protecting file system on public cloud storage. In *2011 International Conference on Cloud and Service Computing*, 2011.

[52] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication As a Threat to the Guest OS. EUROSEC'11, 2011.

[53] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX ATC*, 2018.

[54] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. Last-level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 53–62, New York, NY, USA, 2014. ACM.

[55] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[56] J. Xiong, Y. Zhang, S. Tang, X. Liu, and Z. Yao. Secure encrypted data with authorized deduplication in cloud. *IEEE Access*, 2019.

[57] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Weak Leakage-resilient Client-side Deduplication of Encrypted Data in Cloud Storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.

[58] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, 2013.

[59] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 121–128, Boston, MA, February 2019. USENIX Association.

[60] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.