Sync+Sync: A Covert Channel Built on `fsync` with Storage

Qisheng Jiang   Chundong Wang*
School of Information Science and Technology
ShanghaiTech University, Shanghai, China

Abstract

Scientists have built a variety of covert channels for secretive information transmission with CPU cache and main memory. In this paper, we turn to a lower level in the memory hierarchy, i.e., persistent storage. Most programs store intermediate or eventual results in the form of files and some of them call `fsync` to synchronously persist a file with storage device for orderly persistence. Our quantitative study shows that one program would undergo significantly longer response time for `fsync` call if the other program is concurrently calling `fsync`, although they do not share any data. We further find that, concurrent `fsync` calls contend at multiple levels of storage stack due to sharing software structures (e.g., Ext4's journal) and hardware resources (e.g., disk's I/O dispatch queue).

We accordingly build a covert channel named Sync+Sync. Sync+Sync delivers a transmission bandwidth of 20,000 bits per second at an error rate of about 0.40% with an ordinary solid-state drive. Sync+Sync can be conducted in cross-disk partition, cross-file system, cross-container, cross-virtual machine, and even cross-disk drive fashions, without sharing data between programs. Next, we launch side-channel attacks with Sync+Sync and manage to precisely detect operations of a victim database (e.g., insert/update and B-Tree node split). We also leverage Sync+Sync to distinguish applications and websites with high accuracy by detecting and analyzing their `fsync` frequencies and flushed data volumes. These attacks are useful to support further fine-grained information leakage.

1 Introduction

Computer scientists have explored a variety of covert channels to leak information. CPU cache and memory are main building blocks for many existing covert channels, as programs share them when concurrently running. In this paper, we study a lower layer in the memory hierarchy, i.e., persistent storage and file system, to discover a new covert channel.

The file is the most general form in which programs persistently store their intermediate or eventual execution results. There are different I/O models for programs to consider, including buffered I/O, direct I/O, and synchronous I/O with `fsync` [45, 53, 54, 61, 80]. We focus on synchronous I/O with `fsync` for two reasons. Firstly, many applications employ `fsync` for orderly durability and consistency, such as databases and mail services [57, 61]. Secondly, compared to buffered I/O and direct I/O that write data to the page cache of operating system (OS) and internal device cache of storage device, respectively, `fsync` has a more deterministic and constant timing. In short, when a program calls `fsync` on a particular file, file system (e.g., Ext4) flushes dirty data buffered in memory pages as well as file metadata (e.g., `inode`), for the file through the block I/O (bio) layer to underlying block device, such as a hard disk drive (HDD) or solid-state drive (SSD). To forcefully persist the file, `fsync` carries bio flags (i.e., `REQ_PREFLUSH` and `REQ_FUA`) to instruct storage device to flush the internal cache. Different file systems have different implementations for `fsync`. Take the widely-used Ext4 for example. Ext4 is a journaling file system generally mounted in the default `data=ordered` mode. Upon an `fsync`, it persistently commits file metadata to an on-disk journal after persisting file data [57]. Also, a storage device has mechanisms to guarantee orderly persistence for each `fsync` [13, 80].

With regard to the increasingly large capacity of terabytes or even more per disk drive, multiple programs are likely to run and share the same disk in a local machine or cloud dedicated server [27]. We find that, whenever two programs stay in the same disk partition with the same file system or different disk partitions with different file systems, one of them (*receiver/attacker*) would undergo much longer response time to wait for the return of `fsync` if the other program (*sender/victim*) is concurrently calling `fsync`. For example, with Ext4 mounted on an ordinary SSD, program X has to wait more than twice the time ($\frac{43.13\mu s}{21.39\mu s}$) for the completion of `fsync` when the other program Y is simultaneously calling `fsync`. Note that X and Y operate with absolutely unrelated files, without any on-disk file or in-memory data being shared.

The significant impact of `fsync` is due to contention at multiple levels in the storage stack. When two programs are running within the same file system, they share structures of both file system and storage device. For example, the aforementioned journal of Ext4 is a globally shared structure Ext4 uses to record changes of files in the unit of transactions. The journal is an on-disk circular buffer and transactions must be sequentially committed to it [42, 57, 71]. By default, Ext4 maintains one running transaction to take in modified metadata blocks for files. An `fsync` targeting a file explicitly commits to the journal the current running transaction where the file's `inode` block is placed. The `fsync` may need to wait for the completion of committing previous transaction and also hinder the progress of subsequent transactions. As a result, Ext4 forcefully serializes `fsync`s and each `fsync` must stall until Ext4 commits the transaction for current one. Although researchers have proposed the fast commit to optimize `fsync` [57, 64], our study shows that the introduction of it does not alleviate the substantial interference between `fsync`s. Worse, concurrent `fsync`s also share and compete on other limited software and hardware resources for storage. The bio and disk drivers need to serialize and write down metadata and data for each `fsync` through queues. Consequently, programs running in different disk partitions mounted with different file systems severely suffer from each other's `fsync`s.

These observations motivate us to propose a new covert channel named Sync+Sync for secretive communication. With Sync+Sync, the sender transmits a bit by calling `fsync` or not while the receiver receives the bit by calling `fsync`. For instance, the receiver gets '1' after undergoing a significantly longer response time and receives '0' otherwise. As mentioned, cross-file system and cross-partition Sync+Sync channels are effectual. We also find that receiver and sender staying in different containers and virtual machines (VM) can efficiently communicate with Sync+Sync in cross-container and -VM fashions. To establish a reliable and efficient covert channel, we only need both sides co-located in the same storage device. In addition, Sync+Sync does not demand sender and receiver to share on-disk or in-memory data. Both just call `fsync`s on absolutely unrelated files in the user space, as `fsync` is an unprivileged system call. Our further exploration shows that cross-disk Sync+Sync is also functional. This enables Sync+Sync to gain high flexibility and viability.

In practical, databases widely employ `fsync`s for durability and consistency. Many applications also use `fsync`s to store data. We hence use Sync+Sync to figure out sensitive information from a victim that calls `fsync` over time. For example, we leverage Sync+Sync to identify the runtime operations, such as insert, update, and B-Tree node split, for SQLite [16]. We also differentiate access patterns between common applications and websites with Sync+Sync. One example is that an attacker can leverage Sync+Sync side channel to distinguish Facebook and Twitter with 100% accuracy by analyzing their I/O traces. Sync+Sync hence provides an effective tool to observe a victim's I/O characteristics. Finally, we perform a keystroke attack with Sync+Sync to explore the sensitivity to user inputs with an accuracy of about 99.2%.

Because of the necessity and prevalence of `fsync`s, attackers calling `fsync`s with Sync+Sync are difficult to be discovered. The defense against Sync+Sync is also not straightforward. The performance cost and interference of `fsync` is a classic issue and it is impossible to avoid `fsync`s regarding critical persistence needs of applications. Also, existing techniques to reduce interference for `fsync` such as the aforementioned fast commit for Ext4 are still vulnerable to Sync+Sync. Though, we have given few suggestions to mitigate the impact of Sync+Sync attacks.

To sum up, we make the following contributions.

- We reveal and build a timing-based covert channel named Sync+Sync at the persistent storage with `fsync` system call. To our best knowledge, Sync+Sync is the first covert channel that makes use of `fsync` at the persistent storage without sharing any data between sender and receiver.

- We quantitatively demonstrate that the covert channel of Sync+Sync achieves a transmission bandwidth of 20,000 bits per second (bps) with about 0.40% error rate. Sync+Sync covert channel effectively works in cross-partition, cross-file system, cross-container, cross-VM, and cross-disk fashions. We also introduce various noise to test and justify the robustness of Sync+Sync.

- We perform side-channel attacks to target real-world application programs that use `fsync`s in their implementations. Sync+Sync is able to classify and identify database operations such as insert, update, and B-Tree node split with SQLite. It further differentiates applications and websites from their behaviors of calling `fsync`s with varying frequencies, timings, and data volumes. A keystroke attack is also practicable. Sync+Sync accomplishes all these attacks with high accuracy.

The rest of this paper is organized as follows. We study related works and background knowledge for side-channel attacks in Section 2. We illustrate why `fsync` makes an effectual covert channel in Section 3. We present the attack model and effectiveness of Sync+Sync in communicating between sender and receiver in Section 4. In Section 5, we detail how Sync+Sync is used to perform concrete attacks for information leakage. We show discussions and defenses for Sync+Sync in Section 6. We conclude the paper in Section 7.

2 Background and Related Work

2.1 Side-Channel Attacks

Side-channel attacks pose a significant threat to computer systems and software security. They exploit information leakage through various channels, including shared hardware

resources, system software and hardware, applications, and other observations. In this section, we discuss notable side-channel attacks that are both general and contention-related.

1) Shared Hardware Resources: Shared hardware resources, such as the CPU cache [33, 43, 46, 56, 59, 86], translation lookaside buffer (TLB) [30], branch predictor [21, 22], persistent memory [47, 78], and GPU [1, 51], are common targets for side-channel attacks. Researchers have developed numerous attacks that exploit contention by leveraging these resources. For instance, Prime+Probe [46, 56], Flush+Reload [86], and Prime+Scope [59] utilize cache sets or specific cache lines shared among running programs. Besides CPU cache contention, side channels utilizing other shared resources, such as the directory for cache coherency, cache bandwidth, and coherency states, have also been employed to leak information [84, 85, 87].

2) System Software and Hardware: Side-channel attacks can also target system software and hardware. Lower-level main memory components, such as the OS's page cache and shared page mapping with and without page faults, have been observed to imply side channels [32, 69, 74, 77, 83]. Page cache [27], file systems [6], page walker [89], just-in-time compilation [60], and database queries [31, 39, 44, 67] have been exploited for side-channel attacks too. For example, Gao et al. [27] demonstrated a covert channel across containers by utilizing `sync` to write back all dirty pages in the OS's page cache. Bacs et al. [6] introduced a timing-based side channel called DupeFS that utilized inline file system deduplication. Their observation is that if data an attacker writes have been deduplicated due to a previous write done by the victim, the attacker would observe a shorter response latency.

3) Applications: Web browsers and other applications are also susceptible to side-channel attacks. Researchers have developed attacks that target web browsers by exploiting various vulnerabilities [41, 70, 81]. For example, Kim et al. [41] utilized the interactions between websites and the disk space quota for different websites to infer visited websites, access history, and login status with a particular website.

4) Other Observations: Side-channel attacks have also been launched in other dimensions, such as power consumption analysis [15, 40, 79]. For example, Chen et al. [15] proposed a side channel based on the idle power management for CPUs. By observing the power consumption pattern of a victim, attackers can analyze and deduce sensitive information.

2.2 I/O Stack in OS

Most applications store data in the form of files with file system and underlying persistent storage device, such as SSD or HDD. Without loss of generality, we follow Linux I/O stack shown in Figure 1 to illustrate how OS handles and stores data into block I/O device. The main layers in Linux's storage stack include virtual file system (VFS), file systems, generic block I/O layer, and drivers for specific storage devices.
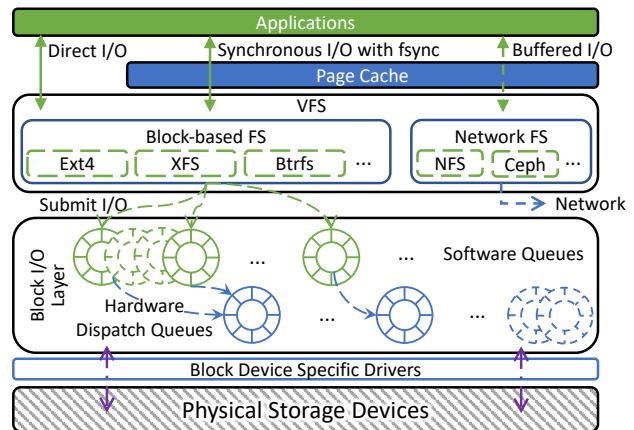

Figure 1: An Overview of Linux I/O Stack.

VFS and File Systems. VFS provides high-level abstractions of file operations for applications to utilize, such as `open`, `close`, `write`, `read`, and `fsync` interfaces. A particular file system (e.g., Ext4 or XFS) handles file operations with underlying storage device. File system is also responsible for organizing and managing storage space and guarantees essential properties like consistency and durability for files. Occasionally, OS may crash due to unexpected events, such as software bugs (e.g., kernel panic) or power outage. File system shall ensure that the modifications onto file metadata and/or data are crash-recoverable in line with a consistency level configured on mounting the file system. Journaling (logging) [42, 57, 71], copy-on-write (CoW) [63], and soft updates [26] are main techniques that file systems adopt to guarantee crash consistency for file metadata (e.g., `inode`) and data. Let us take Ext4 with journaling for illustration because of the wide deployment of it. Ext4 maintains an on-disk journal as a ring buffer. When Ext4 is mounted in the default `data=ordered` mode, only modified file metadata are written to the journal. Ext4 composes a *transaction* in memory with multiple metadata blocks and commits the transaction as a unit to the journal.

Ext4 writes file data and metadata into OS's page cache to handle a write request via buffered I/O. It triggers a write-back of these dirty pages with regard to several conditions. The first one is a periodical flush that occurs every five seconds by default. The second one is that dirty memory pages take more than a proportion of total memory (10% by default) [62]. The third one is an explicit `fsync` received from applications. In the `data=ordered` mode, Ext4 strictly writes file data to on-disk blocks allocated to a file before it commits the file's metadata block to the journal. Later, Ext4 checkpoints metadata blocks in place. The `inode` is the most important metadata for a file and contains the file's length, access time, and access permissions. Once an `inode` block is committed to the journal before a crash happens, Ext4 can recover the file since both file data and metadata have been made durable and retrievable. In addition, Ext4 maintains only one running
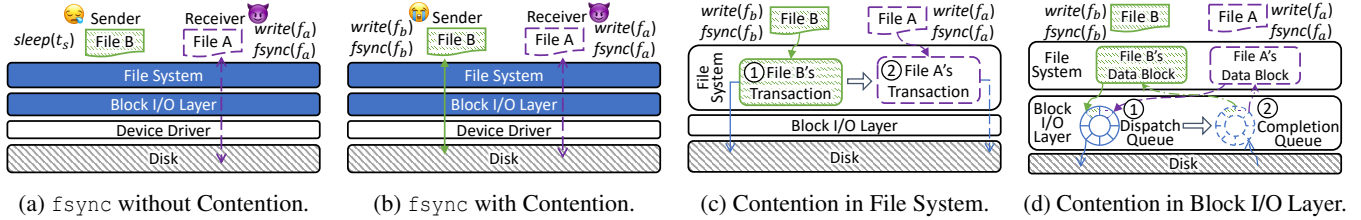
(a) fsync without Contention. (b) fsync with Contention. (c) Contention in File System. (d) Contention in Block I/O Layer.

Figure 2: An Illustration of Contention Caused by fsync.

transaction at runtime. Hence all files concurrently share one transaction. The aforementioned three conditions transform a running transaction to be committing transaction and Ext4 generates the next running transaction. It orderly commits consecutive transactions to the journal.

Block I/O Layer. Block I/O (bio) layer connects a particular file system and underlying storage device. As shown in the middle of Figure 1, Linux maintains multiple software queues (blk-mq) for bio [8]. Each block device contains per-core software staging queues and hardware dispatch queue(s) depending on the device's hardware and driver. File system submits a bio request to software staging queues and waits for I/O scheduler to dispatch. Then the device driver interacts with device to complete the I/O request. Although multiple software queues mitigate competitions between I/O requests at the bio layer, the limited number of hardware queues installed in a block device restricts the device's capability [80]. Thus, concurrent I/O requests issued to a storage device interfere with each other, especially for devices with one single hardware dispatch queue, such as ordinary HDDs and SSDs.

3 The Contention Caused by fsync

OS provides multiple unprivileged system calls, such as fsync, fdatasync, and msync, for user-space programs to synchronously flush data to storage device for orderly persistence. fdatasync and msync are variants of fsync. They either do not flush file metadata or specifically synchronize a memory-mapped file, respectively [53,58]. On receiving an fsync call for a file, file system transfers all modified in-core data and metadata (i.e., dirty pages in OS's page cache) of the file to storage device and issues an ending bio request with two flags (REQ_PREFLUSH and REQ_FUA) set. Then the file's metadata and data are to be forcefully persisted into storage. File system returns a success or fail to the caller program according to the device's returned signal. A successful fsync means that the file's durability is achieved and all changed metadata and data become retrievable even if OS suddenly crashes. Because of the explicit and synchronous persistence of fsync, many applications, particularly ones like databases that are highly concerned with data consistency and durability, widely employ fsyncs in their implementations.

In addition, there is a sync system call that flushes *all*

dirty pages in the OS's page cache to storage device and applications do not need to specify a file [27]. Therefore, sync and fsync are like instructions of flushing all cache lines and one particular cache line, i.e., wbinvd and clflush in the x86 instruction set architecture (ISA), respectively. Since sync causes significantly longer time than fsync and is much easier to be perceived, we focus on fsync in this paper.

3.1 Contention within File System

As shown in Figure 2a and Figure 2b, two programs that call fsyncs interfere each other. Let us use Ext4 for illustration. When a program calls fsync on a particular file, Ext4 flushes data pages and commits the transaction in which the file stays to the journal. As Ext4 maintains only one running transaction and the journal is organized in a ring buffer, the fsync may need to wait for the completion of committing previous transaction. Thus, at the Ext4 file system level, the shared global journal and serialization of committing transactions make one fsync undergo much longer response latency in the presence of another concurrent fsync.

We use the following notations to decompose the fsync latency for a certain file A with and without contention.

$T_{fsync}^A :=$ total fsync latency for the file A.

$T_{data}^A :=$ latency to flush file A' dirty data pages (blocks).

$T_{flush} :=$ latency to flush device cache with flush command.

$T_{meta}(\cdot) :=$ latency to commit a metadata block.

$\Upsilon_{prev} :=$ latency to finish previous committing transaction if applicable.

Assuming that the metadata block of file A is the block τ_n in current transaction, and blocks τ_0 to τ_{n-1} are prior elements for τ_n in the transaction, we have

$$\tau_0 \succeq \tau_1 \succeq ... \succeq \tau_{n-1} \succeq \tau_n, \tag{1}$$

in which $\tau_i \succeq \tau_{i+1}$ means τ_i *happens before* τ_{i+1} (i.e., τ_i has entered the transaction earlier than τ_{i+1}). According to Ext4's fsync behavior, the fsync latency of file A, T_{fsync}^A, is contributed by the following parts,

$$T_{fsync}^A = T_{data}^A + \Sigma_{i=\tau_0}^{\tau_n} T_{meta}(i) + T_{flush} + \Upsilon_{prev}. \tag{2}$$

Table 1: `fsync` Latencies with and without Contention.

| Operation for Measurement	Standalone Latency (*ns*)	Standard Deviation	Standard Error	Operation for Competitor	Contention Latency (*ns*)	Standard Deviation	Standard Error
`ftruncate+fsync`	124725.81	5067.66	506.77	`ftruncate+fsync`	186635.09	4035.34	403.53
				`write+fsync`	165046.60	20210.83	2021.08
				`fsync-only`	165989.67	17878.06	1787.81
`write+fsync`	55707.18	21756.38	2175.64	`ftruncate+fsync`	104521.34	24380.61	2438.06
				`write+fsync`	84345.38	27756.34	2775.63
				`fsync-only`	102094.32	26968.84	2696.88
`fsync-only`	21390.42	2478.57	247.86	`ftruncate+fsync`	47428.24	21527.58	2152.76
				`write+fsync`	51112.34	9088.28	908.83
				`fsync-only`	43133.73	2521.81	252.18

Concurrent `fsync`s cause dramatic impact on `fsync` latencies. For example, we write data to file A without any other program issuing `fsync` and measure the latency as T^A_{fsync1}. Next, we do the same `fsync` on file A but, in the meantime, we make the other program perform `fsync` on a different file B. We denote the second latency for flushing file A as T^A_{fsync2}. As shown in Figure 2c, two programs orderly proceed with Ext4's journaling (①⪰②). Our quantitative tests confirm that T^A_{fsync2} is much greater than T^A_{fsync1}, which is mainly because Υ_{prev} emerges with the interference of syncing file B.

3.2 Contention within Storage Device

`fsync` demands underlying storage device to synchronously write data down and forcefully drain the device cache for eventual persistence. Because of the limited hardware resources of a storage device, further contention occurs at the device level between concurrent `fsync`s. Each `fsync` is transformed to bio requests that orderly flush data and metadata into disk. When an I/O request is submitted to the bio layer, it mainly goes through three stages [5, 8, 80]. **1) Q2I**: The submitted I/O request is preprocessed (e.g., request split and address remapping) and then inserted or merged into a request queue. **2) I2D**: The I/O request waits in the request queue, staying idle until the I/O scheduler dispatches and puts it in the dispatch queue (see ① in Figure 2d). **3) D2C**: The I/O request is issued to corresponding device driver for the device to handle and the bio layer waits at the completion queue for I/O completion (see ② in Figure 2d).

In each stage, I/O requests issued for different files compete for resources against each other. The contention at the D2C stage is even worse due to the limited capability of storage device, as overwhelming synchronous I/O requests are likely to engage I/O scheduler in spending increasingly longer time in dispatching next ones. As shown in Figure 2d, there is a high likelihood of contention for `fsync`s at the device level.

Each `fsync` generates bio requests with `REQ_PREFLUSH` and `REQ_FUA` flags, which are eventually translated to device flush commands to drain the internal device cache. This makes one more contention point between `fsync`s at the device level. For some file system, such as Ext4 or XFS, `fsync`s that do not follow any change of file data or metadata, e.g., with `write` `ftruncate` operations, still interfere with each other, because Ext4 and XFS always issue the device flush command for `fsync` regardless of a modification received or not onto files.
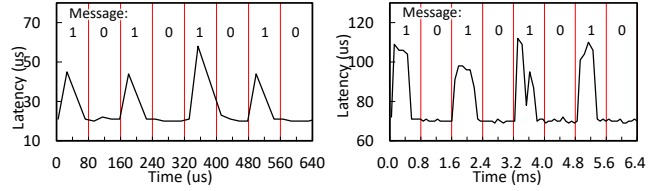
(a) `fsync-only`. (b) `ftruncate+fsync`.

Figure 3: Raw Traces of Cross-file Sync+Sync Channels within Ext4.

3.3 The Viability of `fsync` Channel

In order to empirically understand `fsync`, we test it with and without contention. We make a program X and measure the `fsync` latency without contention under following settings when program X synchronizes a file with Ext4 mounted on an ordinary SSD (SAMSUNG PM883 SATA SSD).

- The `ftruncate` + `fsync` stands for synchronizing a file after truncating the file to a random size, which triggers the commit of file metadata into Ext4's journal.

- The `write` + `fsync` represents that we call `fsync` on a file after overwriting the file for 1KB with data contents.

- The `fsync-only` means that we measure the latency of `fsync` without any modification of file data or metadata.

We only record the latency of `fsync` while the time of writing data or truncating a file is not counted. Next, we keep the other program Y running with the three settings on a different file co-located in the same disk for contention. We measure latencies of program X under different configurations and repeat each configuration for 100 times.

Table 1 captures the average latency for each configuration after 100 executions. It is evident that the latency of `fsync` that program X observes increases significantly when program Y is simultaneously calling `fsync`. For example, the latency of `fsync-only` jumps by 2.2×, 2.4×, and 2.0× when program Y is concurrently doing `ftruncate` + `fsync`, `write` + `fsync`, and `fsync-only`, respectively. As also shown by Table 1, in spite of dramatic increase for latencies, the standard deviation and standard error generally fluctuate in an acceptable and observable range. Among three settings, `write` + `fsync` needs to flush file data to disk, which causes more contention than both `ftruncate` + `fsync` that flushes metadata only and `fsync-only` that aims to retain a file's durability. These three settings are commonly I/O behaviors found in today's applications. Assuming that we make program X probe by calling and measuring `fsync` latency, it can detect whether applications like program Y are calling `fsync`s due to the substantial difference with and without contention. Figure 3 shows two raw traces of varying `fsync` latencies when 1) both programs X and Y are doing with `fsync-only` and 2) both
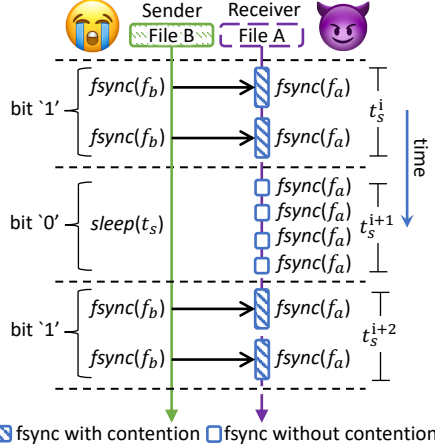
Figure 4: An Illustrative Example of '1' and '0' Transmission Protocol between Sender and Receiver with Sync+Sync.

are doing with `ftruncate` + `fsync`, respectively. It is evident that the latency sensed by program X largely fluctuates due to the impact of concurrent `fsync`s, thereby enabling a clear and reliable channel to transmit data. In all, these qualitative and quantitative observations indicate the viability of building a covert channel with `fsync` at the persistent storage.

4   Sync+Sync Covert Channel

Modern OS utilizes file systems and access control mechanisms to isolate files for users and store data with secrecy and privacy. Cloud vendors enforce further isolation of files for multi-tenants to secretly share a physical machine between containers and VMs. In this section, we focus on leveraging `fsync`s between two entities co-located in a storage device for secretive and reliable communication, regardless of whether they are running in the same OS, different containers, or VMs. As both sides call `fsync` to interact through file system and storage device, we name our new covert channel 'Sync+Sync'. Sync+Sync is timing-based since it transmits information by measuring response time at runtime.

4.1   Attack Model

We assume that there are two co-located entities in a Sync+Sync attack. The receiver (attacker or sink) and the sender (victim or source) concurrently run as user-space programs. Both sides use files to store data. We categorize Sync+Sync attack cases into three classes by their isolation environments. **1) Cross-file**: Sender and receiver access their respective files with exclusive access permissions. These files share the same storage device, residing either in the same disk partition mounted with the same file system (e.g., Ext4) or different partitions mounted with the same or different file systems (e.g., Ext4 and XFS). **2) Cross-container**: Sender

and receiver stay in two containers sharing the same device for their overlay file systems. This is common for hosting multiple containers in a physical machine. For example, by default the Docker [18] stores all containers' overlay data in one directory `/var/lib/docker/overlay2/`. Sender and receiver can also use different directories with different file systems mounted on different partitions. **3) Cross-VM**: Sender and receiver are running in their respective VMs with independent disk images. Again, their image files need to be co-located in one storage device but can stay in different partitions.

With any channel, both sender and receiver can update file size via `ftruncate` + `fsync`, modify their own files via `write` + `fsync`, or keep file synchronized to storage device via `fsync`-only (see Section 3.3). However, by referring to Table 1 and Figure 3, we mainly leverage `fsync`-only to build Sync+Sync channel, because it incurs the shortest latency and implies the highest bandwidth for information transmission.

4.2   Communication Design

To transmit data via any aforementioned channel, the sender conveys bits by calling `fsync`s on a file or not. The receiver continually synchronizes the other file via `fsync` and measures the latency of `fsync` to decide the values of received bits. Figure 4 shows how sender and receiver transmit a bit with the protocol provided by Sync+Sync covert channel. Their pseudocode is shown in Appendix C of the full version [38].

The sender transmits bits via some purposeful file operations. As shown by the left part of Figure 4, in order to transmit a bit '1', the sender synchronizes a file in order to continuously submit I/O requests to disk for a predefined time period t_s named *symbol duration*. Otherwise, the sender sleeps for t_s to transmit a bit '0'. A stream of continuously transmitted bits form a meaningful data frame that is composed of two parts, i.e., header and payload. The header consists of a fixed number of bits with a distinct pattern, used to accurately distinguish the start (boundary) of a frame. The payload is a bit stream with a fixed length and stores actual data that is useful for the receiver.

The receiver receives a bit by measuring `fsync` latencies to detect whether contention happens or not. Before transmission, the sender constantly samples the uninterrupted `fsync` latency to profile a threshold for reference. In a symbol duration during transmission, the receiver checks if `fsync` latency is greater than the profiled threshold. If so, the bit is '1'; otherwise, the bit is '0'. The receiver tracks whether a frame of bits is received by comparing against the header pattern for calibration and synchronization. In case of a match, the receiver extracts payload from the frame.

The threshold to decide '1' or '0' is set empirically, depending on underlying machine's configurations and runtime environmental factors. For example, file system and storage device are being used by many programs which may cause noise with file operations, especially `fsync`s, to affect Sync+Sync. We
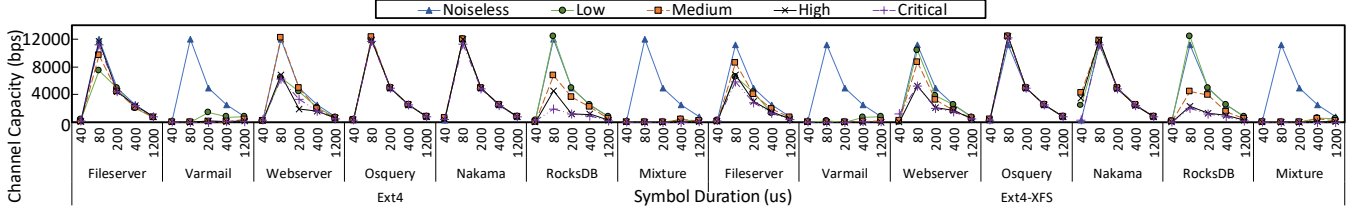
Figure 5: The Capacity of Cross-file Sync+Sync Channel (with and without Noise from Multiple Workloads).

Table 2: Noisy Workloads with Different Settings.

Degree	Filebench (Number of threads)			Osquery	Nakama	RocksDB
	Fileserver	Varmail	Webserver			
Low	1	1	1	Every 60s	0.1 QPS	100:0
Medium	2	2	2	Every 30s	1 QPS	80:20
High	4	4	4	Every 10s	10 QPS	50:50
Critical	8	8	8	Every 1s	100 QPS	20:80

Table 3: Raw Bit Error Rate for Cross-file Covert Channel (Ext4, with and without Noise from Fileserver and Varmail).

Symbol Duration (μs)	Noiseless		Fileserver, 8 Threads		Varmail, 8 Threads	
	$1 \longrightarrow 0$	$0 \longrightarrow 1$	$1 \longrightarrow 0$	$0 \longrightarrow 1$	$1 \longrightarrow 0$	$0 \longrightarrow 1$
40	43.24%	43.24%	44.80%	44.81%	74.12%	74.12%
50	0.40%	0.43%	1.84%	1.87%	67.38%	67.34%
80	0.38%	0.38%	1.40%	1.39%	66.39%	66.37%
200	0.03%	0.03%	1.53%	1.43%	47.63%	47.62%
400	0.00%	0.00%	0.34%	0.34%	41.01%	41.02%
1200	0.00%	0.00%	0.89%	0.89%	37.73%	37.74%

take into account such noise and select a proper threshold in a heuristic approach [15]. In short, we firstly smooth the samples within a symbol duration based on their average. Then, we jointly consider polished latencies from numerous continuous symbol durations to determine the current threshold and periodically update it.

4.3 Performance Evaluation

4.3.1 Evaluation Setup

In order to thoroughly evaluate Sync+Sync channel, we test it on a server with Intel Xeon Gold 6342 CPU and 960GB SAMSUNG PM883 SATA SSD. The OS is Ubuntu 21.04 with kernel 5.15.0-48-generic while the compiler is GCC/G++ 10.3.0. We divide SSD space into few partitions and make different file systems on them, including Ext4, XFS, and Btrfs. We use Docker 20.10.14 to manage containers for which the base image is Ubuntu 21.04. As for cross-VM testing, the guest OS is Ubuntu 21.04 with Ext4 as file system, running with QEMU and Linux Kernel-based Virtual Machine (KVM) version 4.2.1. VM disk images are set in the RAW format with virtio enabled. We use the default writeback cache policy. Each VM is with 8GB DRAM and two vCPUs. Sender and receiver programs are respectively running in two isolated containers (resp. VMs). In all tests, both of them are normal user-space programs and access ordinary files without any privileged permission.

We evaluate three covert channel types, i.e., cross-file, cross-container, and cross-VM, built on different file systems, such as Ext4, XFS, and Btrfs. Due to space limitation, we place a part of results and raw traces in Appendix A of [38]. In the following contexts, the sender and receiver programs only synchronize their respective files using `fsync`, without modifying the files in the cross-file and -container channels. In the case of the cross-VM channel, both the sender and receiver modify the file data before invoking `fsync` on their files.

Besides the noiseless environment, we employ workloads running alongside Sync+Sync to quantitatively assess the robustness of it. As shown in Table 2, we configure each workload with different settings to cause environmental noise at varying degrees. We adopt 1) Filebench [24] to generate Fileserver, Varmail, and Webserver workloads with varying threads, 2) an audit server Osquery [73] with different auditing frequencies, 3) Tsung [52] that simulates 1000 users concurrently sending various requests to a multi-player gaming server Nakama [36] with different Queries Per Second (QPS), and 4) YCSB [17] that issues requests to a prevalent NoSQL database RocksDB [50] with varying Read/Write ratios. Furthermore, we conduct experiments by running all foregoing applications simultaneously (denoted as Mixture).

4.3.2 Channel Capacity

We use the channel capacity to measure the bandwidth of Sync+Sync in bps, which is a theoretical upper bound for Sync+Sync's communication capability. We regard the Sync+Sync channel as a binary symmetric channel for measurement, following the methodology in [28]. The capacity of Sync+Sync (denoted as C) is determined as follows.

$$C = B \times \left[1 - p \log_2\left(\frac{1}{p}\right) - (1-p)\log_2\left(\frac{1}{1-p}\right) \right] \quad (3)$$

where $B = \frac{1}{t_s}$ is the bandwidth and p is the symbol error rate (bit error rate) [68] [1]. We empirically estimate the bit error rate of Sync+Sync channel every ten frames transmitted, each of which carries 8,000 bits. We also vary the symbol duration t_s for each one of three aforementioned covert channels.

---

[1] Sometimes the error rate might be greater 50%, and we still regard channel capacity as 0, since channel with high bit error rate cannot transmit data correctly in our cases.
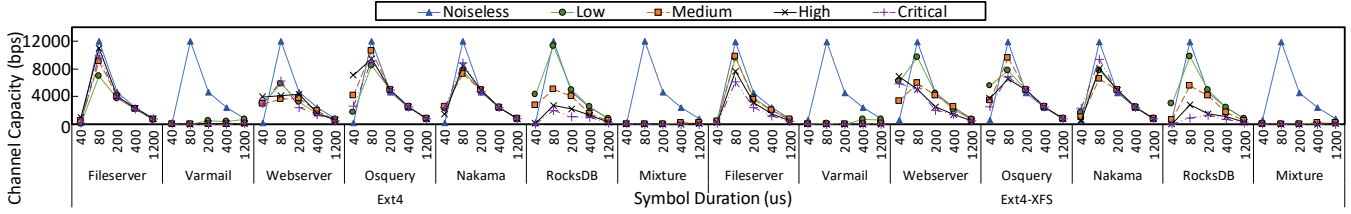
Figure 6: The Capacity of Cross-container Sync+Sync Channel (with and without Noise from Multiple Workloads).
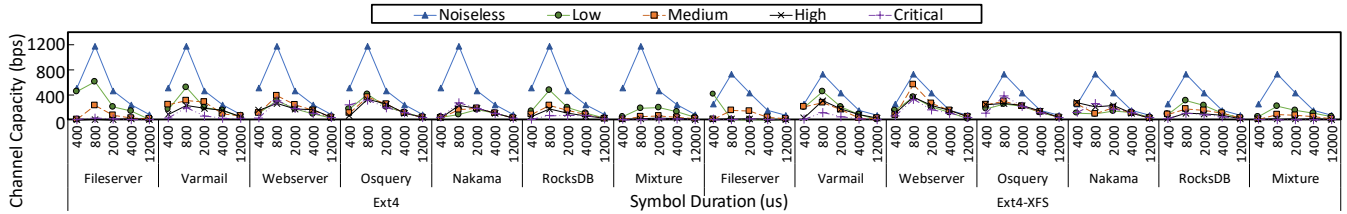

Figure 7: The Capacity of Cross-VM Sync+Sync Channel (with and without Noise from Multiple Workloads).

Intra-partition without noise. Figure 5, Figure 6, and Figure 7 show the capacities of all cross-file, -container, and -VM channels, respectively. To conduct a comprehensive study, we have performed experiments where the sender and receiver are located within the same partition (represented as Ext4 under the X-axis) and different partitions (represented as Ext4-XFS under the X-axis) mounted with different file systems. The cross-file channel, operating within one Ext4 partition, achieves a transmission rate of one bit every $50\mu s$ with an error rate of approximately 0.40% as shown in Table 3. Therefore, Sync+Sync achieves 20,000bps bandwidth without noise. The capacity of the cross-container channel, with two containers co-located in a disk partition with Ext4, is similar to that of the cross-file channel. This can be attributed to the weak isolation provided by the container and overlay file system that heavily rely on the underlying file system. Meanwhile, the capacity of the cross-VM channel is lower, with a maximum bandwidth of about 1200bps. The reason is that the sender and receiver modify their files before invoking fsync in the guest OS to trigger stable contention at the host OS and underlying storage device.

Inter-partition without noise. In all three channels, the receiver is placed in a partition with Ext4, while the sender operates with XFS or Btrfs mounted on another partition of the same SSD. The channels using Ext4 and Btrfs (see Appendix A of [38]) cannot be established with an acceptable bit error rate ($\leq 40\%$). This limitation arises because Btrfs does not flush the device cache for fsync unless there is concrete data modification to the files. In contrast, Ext4 and XFS flush the cache on every fsync, regardless of file modifications. Thus, the capacities of cross-file and -container covert channels based on inter-partition with Ext4-XFS exhibit similar observations with Ext4 in one partition. However, due to the influence of the VM hypervisor, the cross-container channel

gains a much higher transmission frequency (every $80\mu s$) compared to the cross-VM channel (every 12ms). Additionally, the inter-partition cross-VM covert channel experiences higher error rates and lower capacities than the intra-partition channel, mainly due to the impact of partitions. For instance, the capacity with an $800\mu s$ symbol duration of the inter-partition channel is 38.1% lower than that of the intra-partition channel.

4.3.3 Impact of Noisy Workloads

Next, we investigate the influence of different workloads on the Sync+Sync channel in the presence of noise. Our evaluation reveals that most of the workloads introduce noise to all types of Sync+Sync channels with varying degrees, thereby resulting in reduced capacities. For example, the bit error rate shown in Table 3 for cross-file covert channel with $50\mu s$ duration increases to about 1.84% and 67.34% when running Fileserver and Varmail with 8 threads, respectively. As shown in Figure 5, Varmail and RocksDB have the most significant impact on Sync+Sync. This is because both of them frequently invoke fsyncs to ensure the durability of data, especially for Varmail. Fileserver and Webserver also affect the channel capacity, even without fsyncs, particularly with shorter symbol durations. This is due to their consistent file operations (e.g., create and delete) and data writes to multiple files, leading to the accumulation of dirty pages in the OS's page cache and triggering page write-back. Conversely, Osquery and Nakama have limited impact on Sync+Sync, indicating that Sync+Sync is not highly sensitive to auditing activities and network traffic. The capacity under mixed applications (Mixture) is similar to that under Varmail, because Varmail frequently calls fsyncs. In order to mitigate the impact of such noise, we can prolong the channel's symbol duration and adopt fsync with modifying file data to build a more reliable channel. As indicated by Table 3, a longer

symbol duration dramatically decreases the error rate. The results with cross-container channel are similar to those with the cross-file channel (see Figure 6).

As illustrated by Figure 7, the cross-VM covert channel is more susceptible to noise compared to the cross-file and -container channels, even for workloads without `fsync`s. For example, Fileserver introduces severer noise than Varmail. The bit error rate for the cross-VM channel with a symbol duration of 12ms increases from around 0.07% to an average of 33.0% and 16.4% when running Fileserver and Varmail with 8 threads, respectively. We run Fileserver and Varmail in the same disk image under the same guest OS with the sender. Updating multiple files for Fileserver in the guest OS is actually updating the disk image in the host OS, and updated data might not reach the real disk unless calling `fsync`. Once the sender calls `fsync` in the guest OS, the host OS would synchronize the entire disk image file. Therefore, dirty data written by Fileserver are flushed alongside into disk with sender's `fsync`, which causes more difficulties with longer `fsync` latency for receiver to detect due to a larger volume of dirty data. As to Varmail that frequently calls `fsync` to flush every modified email file, each `fsync` is converted to synchronizing the disk image and does not aggregate a lot of dirty data like Fileserver when the sender calls `fsync`s. As a result, the impact of noise caused by Varmail on bit error rate for Sync+Sync is evidently lower than that of Fileserver.

5 Side-Channel Attacks with Sync+Sync

5.1 Database Operations Speculation

Attack Model. In order to perform database operating speculation, we consider a scenario where the victim is a database that stores and accesses data using a disk. The attacker, who is located on the same disk with the victim, operates as an independent process within the same OS, separate container, or VM. Particularly, the attacker does not have permission to access the victim's in-memory data or on-disk files. To infect the victim's environment, the attacker can exploit vulnerabilities via image pollution or social engineering. Due to the compactness of the attacker's code, it is possible for the attacker to inject the victim gadget into useful applications without perception. The attacker has full control over the attacker's program, container, or VM. She/he can manipulate containers or VMs to share the same disk with the victim, similar to co-located allocation [34,75]. Additionally, the attacker can control multiple containers or VMs located on different disks and passively wait for a victim to use those disks. Once both parties share a device, the attacker initiates spying and stealing of sensitive information from the victim using the Sync+Sync channel.

Attack Design. On a disk, we set up isolated files in which a group of them belong to the victim database for data storage while and one file is used by the attacker to detect `fsync` laten-

cies. We utilize SQLite as the victim database, which is widely used and has been exploited for security purposes [54,67,78]. We configure SQLite in the `journal_mode=DELETE` mode and perform various database operations to simulate requests received by the victim. Simultaneously, the attacker invokes `fsync` with three objectives. Firstly, the attacker aims to monitor the rate of insert and update requests in the victim database (Section 5.1.1). Secondly, the attacker aims to detect internal structural changes in the victim database, such as a node split in the B-Tree used for indexing (Section 5.1.2). Thirdly, the attacker aims to identify and resemble a sequence of database operations executed by the victim, thereby extracting more fine-grained information (Section 5.1.3).

5.1.1 Insert/Update Ratio over Time

Insert and update requests are tightly correlated to the `fsync` latency, because when handling such requests, databases such as SQLite utilize redo or undo logging with `fsync`s to ensure consistency and durability. We adopt Mobibench [20,37] to repeatedly insert or update primary keys and corresponding data into the database, with the idle period adjusted for each insert to demonstrate different request frequencies. By leveraging the Sync+Sync side channel, the attacker detects longer `fsync` latencies when the victim database synchronizes files. The attacker counts latency samples above a threshold (50μs in our evaluation) that has been determined through profiling. The attacker accordingly estimates the victim's insert and update activities and calculates the rate of insert/update over time. We repeat this attack for ten times, each in 30 minutes.

Figure 8a depicts the relation between the number of insert requests per minute for SQLite and the number of samples above the threshold, using a logarithmic scale for the Y-axis. The number of samples above the threshold is approximately ten times higher than the number of requests per minute. The reason for such an order of magnitude difference is twofold. Firstly, SQLite requires multiple `fsync`s to flush log and data files in order to commit a single transaction. Secondly, the attacker's `fsync` latency is shorter than that of the victim because the victim flushes both data and metadata for database files to complete a database transaction. The attacker needs to call `fsync` for multiple times to cover the entire course of a database transaction. Consequently, the attacker is able to identify when the victim handles an insert or update request and thus calculate the rate of such requests over time. In a long run, the attacker can figure out the victim's workload characteristics, such as the insert/update frequency, modified data per request, and peak/non-peak periods in a day.

5.1.2 B-Tree Split Detection

SQLite utilizes an on-disk B-Tree for indexing keys. Each B-Tree node has a limited size. A fully filled node triggers a split that results in two nodes. SQLite calls `fsync`s to persist

(a) SQLite Insert Requests Estimation.　(b) Raw Trace for Insert with or without Split.　(c) Estimated Latency Distribution of SQL.
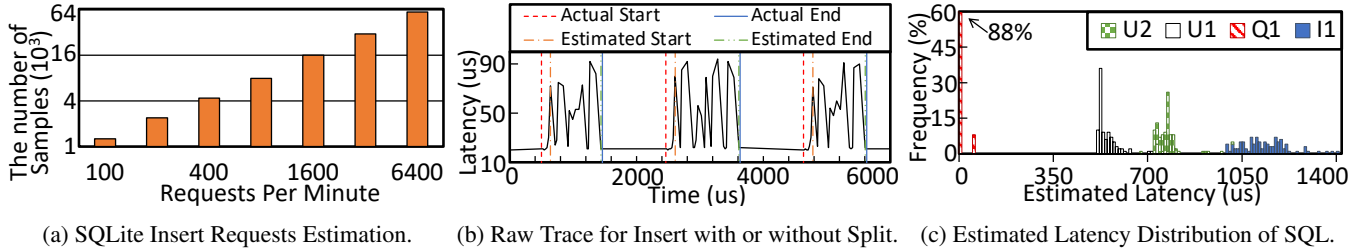
Figure 8: An Illustration of Sync+Sync Side Channel for Databases.

both new nodes to ensure consistency and durability. Under a relatively consistent workload, a node split leads to a longer committing latency for an insert into the B-Tree compared to inserts without node splits. Hence, the attacker can detect such structural changes in the SQLite database via Sync+Sync side channel. For example, the raw trace in Figure 8b demonstrates three different insert requests. The latency of middle one experiences longer than latencies of others due to the occurrence of a node split, while the other two have not involved two nodes to be persisted via `fsync`s. Additionally, researchers have mentioned that sensitive data stored in a B-Tree-based database could be leaked by exploiting node splits [25].

The attacker does not have knowledge of when an insert request starts or ends but can only determine when the corresponding `fsync` latencies spike and return to normal. Consequently, the attacker has introduced an estimated committing latency to speculate whether an insert with a node split has occurred. The attacker considers the first sample time with an `fsync` latency greater than a threshold as the estimated start timestamp for an insert request. The attacker detects the last sample that exceeds the threshold and estimates an end time, which is calculated by adding the measured `fsync` latency to the timestamp of the last sample. Note that the idle period before the next insert request is expected to be sufficient for the attacker to determine if a detected `fsync` belongs to the current insert or not. Furthermore, recent studies have shown that typical workloads in commercial environments are dominated by inserting small values (less than 1KB or 100B), which are unlikely to flush more than a 4KB block [10, 82]. Therefore, the impact of a B-Tree node split on two 4KB blocks with `fsync`s is realistically substantial.

The estimated start time and end time for an insert are exemplified in Figure 8b. The estimated committing latency for the victim database exhibits an identical observation to the actual committing latency. We insert 400 different primary keys and data into SQLite and cause 49 node splits in all. We set the threshold at $70\mu s$ to estimate the latency for each insert, and empirically classify an insert with an estimated committing latency greater than $1000\mu s$ as one causing a split. The attacker successfully detects 43 node splits, achieving an accuracy of 87.8%, and the F1-score is 0.84. By leveraging more sophisticated algorithms such as learning-based methods, the attacker may have even higher capability to dif-

ferentiate inserts with node splits from normal inserts. This is yet not the focus of building Sync+Sync side channel in this paper.

5.1.3 Database Operation Leakage

By utilizing `fsync` latency as a probe, we can further categorize various database operations to discover and learn about the operations executed by the victim. In this study, we create database tables from the NPPES dataset [11, 78] containing two tables. One table is a `basic` that records users' basic information (e.g., their full names), using the National Provider Identifiers (NPI) as primary keys. The other table named `location` stores users' addresses, including city and state, with the NPI serving as a foreign key for the `basic` table.

Listing 1 provides examples of four database operations that the attacker can classify using the Sync+Sync side channel. I1 represents the insertion of 1000 records, while Q1 performs queries to count records. U1 updates a single record in the `basic` table. Comparatively, U2 updates 1000 records in the `location` table. We repeat the U1 and U2 operations for 100 times and estimate their `fsync` latencies using a similar way as presented in Section 5.1.2, with a threshold of $50\mu s$ based on profiling. Figure 8c shows the distribution of estimated latencies for each database operation. For the Q1 operations, the majority of estimated latencies are close to 0 because queries do not involve `fsync` to flush data. As to update operations (U1 and U2), the different numbers of records being modified result in varying data volumes for `fsync`s, leading to different distributions of estimated latencies. Whereas, the insert operations (I1) exhibit latencies that are consistently higher than other operations, even though I1 and U2 involve the same number of records. The reason is that, in contrast to an update that modifies one node only, some insert may cause node split that leads to longer latency.

Taking estimated latency for each operation as its feature, we employ the classic k-nearest neighbors (k-NN) algorithm [4] for classification. We calculate the Euclidean distances between a newly detected operation and each of known operations. Then, we assign a type to the new operation based on the majority of the k nearest neighbors. We randomly choose 70% samples for each operation as training set and the rest 30% samples are used as the testing set

to evaluate the k-NN algorithm. In the end, 115 out of 120 operations are correctly classified, resulting in an accuracy of 95.8% for the Sync+Sync side channel. The F1-scores for `I1`, `Q1`, `U1`, and `U2` are 0.98, 0.98, 0.93, and 0.95, respectively.

```
1  -- I1: Insert 1000 records in 'basic' table
2  INSERT INTO basic values (...), (...);
3  -- Q1: Count records in 'location' table
4  SELECT COUNT(*) FROM location WHERE lower(city)=
     'Anaheim';
5  -- U1: Update 1 record in 'basic' table
6  UPDATE basic SET name = 'new name' WHERE npi ==
     20230809;
7  -- U2: Update 1000 records in 'location' table
8  UPDATE location SET city = 'new city' WHERE npi
     == 20230809;
```

Listing 1: Classified SQL examples.

5.1.4 Comparison

The way Sync+Sync leaks information is not only effectual and portable, but also difficult to be perceived. For instance, Chen et al. [15] proposed a side channel based on idle power management for CPUs, allowing them to spy on a victim's network traffic, such as HTTP traffic load measured in requests per minute. However, the information they obtain is more coarse-grained compared to what is achieved with Sync+Sync that provides a detailed trace of the victim database's operations (insert/query/update). Moreover, Sync+Sync is built on `fsync`, which is commonly and frequently used by applications on various platforms. By contrast, the side channel presented by Chen et al. [15] relies on the prerequisite condition that CPU enters an energy-saving mode and switches back. The ND2DB attack [25] detects B-Tree node splits by measuring the response time of an insert request. However, the response time can be easily influenced by user software overhead and network latency. In our empirical study, using a threshold of $1300\mu s$, the ND2DB attack detects only 75.5% of node splits with an F1-score of 0.65. Whereas Sync+Sync detects 87.8% of node splits with an F1-score of 0.84. The NVLeak attack [78] can figure out the database operations like Sync+Sync and achieves an 84% classification accuracy, which is yet much lower than the 95.8% accuracy of Sync+Sync. Furthermore, the NVLeak attack only works with Optane memory that Intel has winded down [3, 90]. Additionally, Sync+Sync can distinguish different database operations and may jointly make use of other techniques to inflict more fine-grained leakage with the victim database [31, 39, 44, 67]. In all, the prevalence of `fsync` entitles high efficacy, portability, and unnoticeability to Sync+Sync.

5.2 Application Information Leakage

Attack Model. We assume a scenario where the victim is an application (e.g., Linux/Android application or web browser [65, 66]) that accesses files in a disk which the attacker shares with the victim when running in the other process, container, or VM. Again the attacker neither has access permission with the victim's files nor shares data with the victim. To leak information from the application, the attacker synchronizes her/his file to record `fsync` latencies every $40\mu s$ based on profiling and spies on the victim's I/O behaviors.

Attack Design. To perform information leakage for applications, we utilize Mobibench [20] to replay I/O traces for victim applications. Each I/O trace consists of a series of I/O-related system calls such as `read`, `write`, and `fsync`. We execute an I/O trace to simulate the corresponding application's behaviors while the attacker is simultaneously calling `fsync`s. In practical, different applications exhibit varying calling patterns of `fsync`s in terms of frequency and data volume to be flushed. These variations result in different `fsync` latencies sensed by the attacker. Sync+Sync is thus able to distinguish different applications. It also manages to fingerprint websites with web browsers under certain conditions.

Application Fingerprinting. We firstly run I/O traces of Twitter and Facebook Android applications, provided by Mobibench, as two victims separately for 100 times. We record and show the attacker's `fsync` latency distribution in Figure 9. It is evident that Twitter and Facebook exhibit distinctly different curves in each of their 100 runs. For instance, the number of samples between $50\mu s$ to $100\mu s$ for Facebook is significantly higher than that of Twitter. To accurately classify I/O traces for different applications, we utilize the latency distribution and Euclidean distances as features and metrics, respectively, for the k-NN algorithm. Similar to the k-NN mentioned in Section 5.1.3, we consider the majority of the k nearest neighbors for an application trace as the application's type. We randomly divide each application trace into 70% and 30% samples for training and testing, respectively. In the evaluation, we have 60 test cases ($100 \times 30\%$ for each application), and Sync+Sync correctly categorizes all of them.

Website Fingerprinting. Next, to check if Sync+Sync can distinguish different websites based on their `fsync` usage patterns, we create an evaluation dataset as follows. Firstly, we randomly select 20 websites from the Alexa Top 100 websites. Then, with the Chrome web browser (Version 113 with default settings) [29] running on a computer installed with Ubuntu 22.04, we visit the front page of each website as the victim. Simultaneously, we capture the I/O trace of each
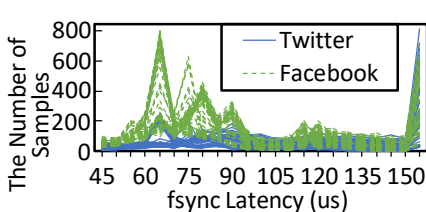
Table 4: Classification Accuracy of Sync+Sync for Websites.

Website	Average # fsync	Accuracy	F1-score	Website	Average # fsync	Accuracy	F1-score
360.cn	10.6	3.3%	0.04	imdb.com	16.1	13.3%	0.19
adobe.com	11.5	0.0%	0.00	jd.com	14.9	56.7%	0.65
amazon.com	16.2	13.3%	0.08	live.com	10.6	13.3%	0.14
apple.com	11.5	16.7%	0.17	microsoft.com	12.1	3.3%	0.04
baidu.com	14.4	6.7%	0.03	qq.com	264.6	100.0%	1.00
bing.com	15.0	6.7%	0.08	sina.com.cn	40.8	96.7%	0.98
booking.com	15.9	0.0%	0.00	sohu.com	14.4	46.7%	0.44
cnn.com	15.2	6.7%	0.10	taobao.com	10.6	23.3%	0.16
detik.com	10.1	10.0%	0.13	tmall.com	11.4	3.3%	0.03
github.com	12.1	13.3%	0.18	yahoo.co.jp	11.4	6.7%	0.06

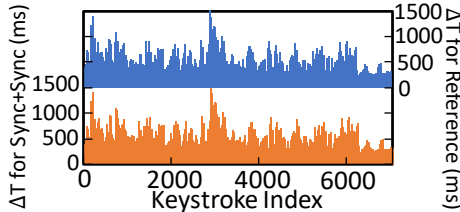Figure 9: `fsync` Latency Distribution for I/O Traces of Twitter and Facebook in Application Fingerprinting.

Figure 10: The Inter-keystroke Timings (ΔT) from the Reference Typer (Top, Right Y axis) and the Sync+Sync (Bottom, Left Y axis).

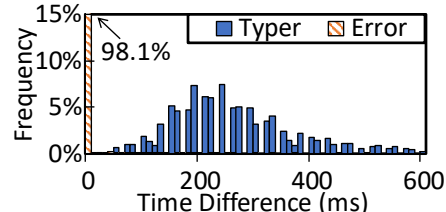Figure 11: The Time Distribution of the Reference Typers Compared to the Error Distribution of the Sync+Sync.

website using strace [14] for 5 seconds, which is sufficient for loading a webpage. We repeat this procedure for 100 times with each website and overall collect 2000 I/O traces.

With these I/O traces, we record the attacker's `fsync` latency distribution when replaying each I/O trace and classify them using the k-NN algorithm with the same setup as application fingerprinting. Table 4 shows the classification accuracy and F1-score for each website, as well as the average number of `fsync`s invoked by each website. Most websites do not commonly use `fsync` and hence have similar `fsync` usage patterns. As a result, it is challenging for Sync+Sync to distinguish them from each other. However, some websites, such as qq.com and sina.com.cn, invoke `fsync` more frequently and exhibit different I/O behaviors. Therefore, Sync+Sync recognizes these websites at high accuracies of, for example, 100% and 96.7% for qq.com and sina.com.cn, respectively. Our analysis shows that, to persistently store data, both websites use the Indexed Database [2] that most of the browsers provides [7, 23, 49, 55]. As mentioned, database relies on `fsync` to ensure data consistency and durability. This explains why Sync+Sync successfully fingerprints websites that frequently synchronizes data with the Indexed Database.

Comparison. Distinguishing (fingerprinting) applications and websites enables an attacker to infer which application a victim is using or what website a browser is displaying, causing serious breach of user privacy [9, 35, 41, 76]. Sync+Sync shares similarities with the attack proposed by Kim et al. [41], as they both exploit storage to fingerprint victims. However, they differ in the explored observations. Kim et al. [41] use the disk space quota demanded by a web browser for each website's temporary storage, while Sync+Sync is based on different `fsync` usage patterns. Website fingerprinting attack conducted by Kim et al. [41] achieves a 97.3% inference accuracy, whereas Sync+Sync's accuracy varies depending on the characteristics of websites. Additionally, the side channel studied by Kim et al. [41] only applies to browsers, while Sync+Sync is widely applicable to applications that call `fsync`.

5.3 Keystroke Attack

Attack Model. We assume that the victim is entering user input either locally or remotely through a network connection. Each keystroke is then transmitted to a service program, which stores the input on an SSD similar to the attack studied by Liu et al. [47]. For every keystroke typed by the victim, the service program auto-commits the user input by storing it in a file with an `fsync`, in order to persistently track the user's latest input. The attacker and the service program are co-located to be sharing the same disk. However, due to OS-level isolation, any direct communication between them is not possible. Also the attacker has no access permission to the victim's and the service program's files or share any data with the latter two.

Attack Design. To conduct a keystroke attack, we utilize the Keystroke100 dataset [48]. It contains inter-keystroke latencies from 100 different typers who entered the same eight-letter password, 'try4-mbs', ten times each, resulting in a total of 7,000 inter-keystroke timings. In our attack scenario, the victim sends keystrokes to the service program with the corresponding delays of inter-keystroke timings prerecorded in [48]. The service program receives the user input and stores it using `fsync`. In the meantime, the attacker performs the Sync+Sync attack by continuously invoking `fsync` to infer whether a user input is stored by measuring the attacker's `fsync` latency. Given a stored user input, the attacker can detect an increased `fsync` latency. Conversely, if the `fsync` latency remains low, the attacker deduces with a high probability that no user input has been sent to the service program. In our evaluation, we set a threshold of 54μs through profiling to distinguish if a user input is received and stored.

To assess the accuracy of Sync+Sync, we calculate the timing difference between the ground-truth latencies from the prerecorded dataset and detected ones. Figure 10 illustrates a back-to-back comparison of the inter-keystroke timings between the reference typers and Sync+Sync attack. The difference is negligible, as Sync+Sync successfully detects keystrokes with an accuracy of 99.2%. Sync+Sync may misjudge keystrokes due to noise and less intense contention. Figure 11 presents the error distribution of Sync+Sync side channel in comparison to the timing distributions of the ground

Table 5: Raw Bit Error Rate with and without Fast Commit.

Symbol Duration (μs)	Sender: ftruncate + fsync Receiver: fsync-only		Sender: ftruncate + fsync Receiver: ftruncate + fsync	
	Fast Commit	Normal Commit	Fast Commit	Normal Commit
200	0.09%	0.15%	15.93%	23.75%
400	0.06%	0.07%	2.65%	2.59%
800	0.01%	0.17%	0.16%	0.12%
1600	0.06%	0.03%	0.03%	0.10%
2400	0.04%	0.10%	0.14%	0.06%

Figure 12: Transaction Average and Tail Latencies for SQLite in VM with and without DoS Attack.

truth. On average, the error in received timings for Sync+Sync side channel is 2.5ms, with 98.1% of errors being less than 10ms. This further justifies the capability of Sync+Sync since an inter-keystroke latency is generally no less than 100ms.

Comparison. Inter-keystroke timing has been widely considered in software-based side-channel attacks. It allows attackers to reveal sensitive information through simple statistical techniques using keystroke timings [15, 72]. For instance, Song et al. [72] demonstrate that attackers can uncover information about the keys typed by analyzing users' typing patterns to recover passwords entered during SSH connections. Unlike Gruss et al. [32] that detect a keystroke when the OS's page cache loads related pages upon a key input issued by the user, Sync+Sync functions at the persistent storage with a hypothesis that fsync operations are needed to store user input keys. Sync+Sync is hence infeasible for applications that do not auto-commit and invoke fsync on the arrival of user input. With precise keystroke timings, Sync+Sync can jointly work with techniques like machine learning to guess passwords or infer written characters [15, 19, 47, 72, 88].

Recently Chen et al. [15] performed a keystroke attack by measuring the uncore power status, as the network traffic and encryption stack of SSH connections affect a system's uncore power. They achieved an F1-score of 0.93 with an error rate of 4.9%. Later Liu et al. [47] did a similar keystroke attack on Intel Optane persistent memory (PMEM). They persistently store the user input to a key-value store on PMEM after every keystroke typed by the victim and detect inter-keystroke timings by probing the Read-Modify-Write (RMW) buffer of PMEM to distinguish RMW hits from misses. They achieved an F1-score of 0.99 with an error rate of 1.04%. Comparatively, Sync+Sync achieves not only an F1-score of 0.996, but also with a lower error rate of 0.81%.

6 Discussions

In this section, we further study the impact of fast commit in Ext4, DoS attack, and the defense mechanisms about Sync+Sync. In Appendix B of [38], we also explore the cross-disk Sync+Sync channel, the channel on other platforms and NVMe SSD, and the impact of Pass-through Disk in VM.

The Impact of Fast Commit in Ext4. Fast commit is a new feature introduced in Ext4 to eliminate unrelated data when invoking fsync so as to reduce contention for fsyncs [57, 64]. We test if Sync+Sync still functions with fast
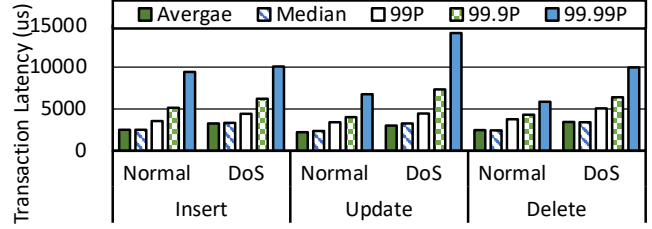
commit or not. To trigger Ext4 journaling every fsync, sender utilizes ftruncate to change file size randomly and invokes fsync to enable Sync+Sync channel, i.e., ftruncate+fsync. Receiver works with ftruncate+fsync or fsync-only. Table 5 presents bit error rates for Sync+Sync channel with and without fast commit. Evidently fast commit does not affect Sync+Sync, since at the same symbol duration Sync+Sync yields identical bit error rate despite the use of fast commit.

DoS Attack. The interference between fsyncs impairs performance and implies the potential of DoS attack for Sync+Sync. As VMs isolate programs with hypervisor handling I/Os for them, the difficulty of launching DoS attacks is higher than doing so with programs co-located in the same OS. We run two VMs with independent disk images. The attacker in malicious VM continually writes its file and invokes fsyncs. We utilize Mobibench to issue Insert/Update/Delete workloads with SQLite for 100,000 times in the victim VM. As shown in Figure 12, the throughput of SQLite degrades by 22.5%, 25.9%, and 28.4% with Insert, Update, and Deletion requests, respectively. The 99.9P (99.9th percentile) tail latency increases by 20.8%, 82.2%, and 47.7%, respectively. These justify Sync+Sync's capability in making DoS attacks.

Defense Mechanisms. It is impracticable for applications to avoid using fsyncs. One straightforward way to defend against Sync+Sync attacks is to prevent applications from being co-located with other users/applications in the same machine, especially on the same storage device. Yet such hardware isolation increases the operating cost with extra devices or even machines and causes waste of disk space.

Using network-based distributed file systems (e.g., Ceph [12]), to host applications remotely helps to mitigate the effect of Sync+Sync, as fsync operations become more sophisticated. Firstly, fsync latency is not only influenced by the storage devices, but also affected by the network. Secondly, distributed file systems often deploy replica in different physical machines to provide data consistency and durability, which increases difficulty of generating contention for adversaries to detect. However, this may not be feasible for some applications that are sensitive to network delay and need local storage to provide high performance. Such applications can consider utilizing a background thread that randomly calls fsyncs to create noise for Sync+Sync, since Sync+Sync is

sensitive to noise. However, to blur `fsync` latencies by introducing extra `fsync`s may incur performance penalty.

7 Conclusion

In this paper, we build a new Sync+Sync side channel that entails a practical concern for programs that are co-located in persistent storage and call `fsync`s for durability. For example, Sync+Sync on Ext4 and an ordinary SSD establishes a covert channel with 20,000bps bandwidth at about 0.40% error rate. Sync+Sync is made effectual by the contention between concurrent `fsync`s on sharing file system's structures and storage device's hardware resources. Extensive experiments show that, leveraging Sync+Sync we manage to launch concrete attacks, such as precisely detecting operations of victim database, distinguishing applications, and fingerprinting websites. We have verified the viability of Sync+Sync on various platforms including Linux, Windows, and MacOS, and responsibly disclosed Sync+Sync to the security teams of Linux, Microsoft, and Apple. We hope Sync+Sync could encourage researchers to further study side-channel information leakage at the persistent storage.

Acknowledgement

References

[1] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. Network-on-chip microarchitecture-based covert channel in GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 565–577, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Ali Alabbas and Joshua Bell. Indexed database api 3.0. https://www.chromium.org/developers/design-documents/indexeddb, 2023.

[3] Paul Alcorn. Intel kills Optane memory business, pays $559 million inventory write-off. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good, August 2022.

[4] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[5] Jens Axboe, Alan D. Brunelle, and Nathan Scott. blktrace(8) - linux man page. https://linux.die.net/man/8/blktrace, 2006.

[6] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. DUPEFS: Leaking data over the network with filesystem deduplication side channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 281–296, Santa Clara, CA, February 2022. USENIX Association.

[7] Kayce Basques. View and change IndexedDB data. https://developer.chrome.com/docs/devtools/storage/indexeddb/, 2023.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 605–616, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST'20, page 209–224, USA, 2020. USENIX Association.

[11] Centers for Medicare & Medicaid Services. NPPES dataset. https://download.cms.gov/nppes/NPI_Files.html, December 2019.

[12] Ceph Open Source Community. Ceph file system — ceph documentation. https://docs.ceph.com/en/quincy/cephfs/index.html, 2016.

[13] Yun-Sheng Chang and Ren-Shuo Liu. OPTR: Order-preserving translation and recovery design for SSDs with a standard block device interface. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1009–1023, USA, 2019. USENIX Association.

[14] Vitaly Chaykovsky. linux syscall tracer. https://strace.io/, 2023.

[15] Paizhuo Chen, Lei Li, and Zhice Yang. Cross-VM and Cross-Processor covert channels exploiting processor idle power management. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 733–750, Virtual Event, August 2021. USENIX Association.

[16] SQLite Consortium. SQLite. `https://www.sqlite.org/index.html`. Accessed: 2022-8-31.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Docker Inc. Docker overview. `https://docs.docker.com/get-started/overview/`, February 2023.

[19] Kwesi Elliot, Jonathan Graham, Yusef Yassin, Trenton Ward, John Caldwell, and Tawab Attie. A comparison of machine learning algorithms in keystroke dynamics. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 127–132, 2019.

[20] ESOS-Lab. Mobile benchmark tool (mobibench). `https://github.com/ESOS-Lab/Mobibench`, May 2020.

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[22] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 693–707, New York, NY, USA, 2018. Association for Computing Machinery.

[23] David Fahlander. IndexedDB on Safari. `https://dexie.org/docs/IndexedDB-on-Safari`, 2023.

[24] Filebench. Filebench: File system and storage benchmark that uses a custom language to generate a large variety of workloads. `https://github.com/filebench/filebench`, February 2020.

[25] Ariel Futoransky, Damián Saura, and Ariel Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, USA, 2007. USENIX Association.

[26] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, may 2000.

[27] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. Houdini's escape: Breaking the resource rein of Linux control groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1073–1086, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Steven Gianvecchio, Haining Wang, Duminda Wijesekera, and Sushil Jajodia. Model-based covert timing channels: Automated modeling and evasion. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, page 211–230, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Google. Google Chrome - download the fast, secure browser from Google. `http://google.com/chrome`, 2023.

[30] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 955–972, USA, 2018. USENIX Association.

[31] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 315–331, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 167–180, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1458–1473, 2022.

[34] Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Trans. Dependable Secur. Comput.*, 14(1):95–108, jan 2017.

[35] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, Austin, TX, August 2016. USENIX Association.

[36] Heroic Labs. Nakama server. https://heroiclabs.com/docs/nakama/getting-started/index.html, 2023.

[37] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android storage performance analysis tool. In Stefan Wagner and Horst Lichter, editors, *Software Engineering 2013 - Workshopband*, pages 327–340, Bonn, 2013. Gesellschaft für Informatik e.V.

[38] Qisheng Jiang and Chundong Wang. Sync+Sync: A covert channel built on fsync with storage. *arXiv preprint arXiv:2309.07657*, 2023.

[39] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1329–1340, New York, NY, USA, 2016. Association for Computing Machinery.

[40] S. Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R. Karpuzcu. POWERT channels: A novel class of covert communication exploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 291–303, 2019.

[41] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 410–421, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-Journal: Scalable Per-Core journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 893–906. USENIX Association, July 2021.

[43] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2020.

[44] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314, 2018.

[45] Wongun Lee, Keonwoo Lee, Hankeun Sun, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 235–247, USA, 2015. USENIX Association.

[46] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[47] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-channel attacks on optane persistent memory. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, August 2023.

[48] Chen Change Loy. Keystroke100 dataset. http://personal.ie.cuhk.edu.hk/~ccloy/downloads_keystroke100.html, March 2014.

[49] MDN contributors. Using IndexedDB. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB, 2023.

[50] Meta Platforms, Inc. RocksDB. https://rocksdb.org, 2022.

[51] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Nicolas Niclausse. Tsung. http://tsung.erlang-projects.org, 2017.

[53] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1–14, USA, 2006. USENIX Association.

[54] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction support in log-structured filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 345–362, Santa Clara, CA, February 2022. USENIX Association.

[55] Jeremy Orlow. IndexedDB design doc. https://www.chromium.org/developers/design-documents/indexeddb, 2023.

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[57] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, Santa Clara, CA, July 2017. USENIX Association.

[58] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 225–238, New York, NY, USA, 2013. Association for Computing Machinery.

[59] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2906–2920, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. DeJITLeak: Eliminating JIT-induced timing side-channel leaks. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 872–884, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[62] Red Hat. 5.3.9.5. /proc/sys/vm/. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/4/html/reference_guide/s3-proc-sys-vm, 2023.

[63] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[64] Marta Rybczyńska. Fast commits for Ext4. https://lwn.net/Articles/842385/, January 2021.

[65] Iskander Sanchez-Rola, Davide Balzarotti, Christopher Kruegel, Giovanni Vigna, and Igor Santos. Dirty clicks: A study of the usability and security implications of click-related behaviors on the web. In *Proceedings of The Web Conference 2020*, WWW '20, page 395–406, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1502–1514, New York, NY, USA, 2018. Association for Computing Machinery.

[67] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *30th USENIX Security Symposium (USENIX Security 21)*, San Diego, CA, August 2021. USENIX Association.

[68] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[69] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. Association for Computing Machinery.

[70] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2863 – 2880, San Diego, CA, August 2021. USENIX Association.

[71] Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 227–240, USA, 2018. USENIX Association.

[72] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, USA, 2001. USENIX Association.

[73] The Linux Foundation. "osquery. https://www.osquery.io, 2023.

[74] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th*

USENIX Conference on Security Symposium, SEC'17, page 1041–1056, USA, 2017. USENIX Association.

[75] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 913–928, USA, 2015. USENIX Association.

[76] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 143–157, USA, 2014. USENIX Association.

[77] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[78] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. NVLeak: Off-chip side-channel attacks via non-volatile memory systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, August 2023.

[79] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 393–406, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 211–226, USA, 2018. USENIX Association.

[81] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. Rendering contention channel made practical in web browsers. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3183–3199, Boston, MA, August 2022. USENIX Association.

[82] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-Trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 71–82, USA, 2015. USENIX Association.

[83] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[84] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904, 2019.

[85] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179, Vienna, Austria, February 2018. IEEE Press.

[86] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[87] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 346–367, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[88] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th Conference on USENIX Security Symposium*, page 17–32, USA, 2009. USENIX Association.

[89] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Binoculars: Contention-based side-channel attacks exploiting the page walker. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 699–716, Boston, MA, August 2022. USENIX Association.

[90] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-file virtualization for userspace persistent memory filesystems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, page 1–15, USA, February 2023. USENIX Association.