File Notification Attacks: Templating and Exploiting Side-Channel Leakage from the File-Notification Systems on Linux, Windows, and macOS

Sudheendra Raghav Neela
Graz University of Technology
Austria

Xufan Zhao
Graz University of Technology
Austria

Jeanette Angelika Wultsch
Graz University of Technology
Austria

Hannes Weissteiner
Graz University of Technology
Austria

Florian Draschbacher*
Graz University of Technology
Austria

Stefan Gast
Graz University of Technology
Austria

Daniel Gruss
Graz University of Technology
Austria

Abstract

Modern operating systems like Linux, Windows, and macOS provide built-in subsystems to monitor filesystem events: `inotify`, `ReadDirectoryChangesW`, and `FSEvents`. User processes can subscribe to receive file-operation notifications when actions like accessing, writing, opening, and closing are performed on a monitored file or directory. However, the attack surface opened up by these notification systems is not well understood, as prior work only showed that file-open events leak application startups on Android.

In this paper, we demonstrate the first generic file-notification template attacks on Linux, Windows, and macOS, allowing to spy on various system and user activity. The foundation of our attack is a new approach to discover exploitable file-notification side-channel leakage. We template the broad range of cross-user leakage from various operations including running terminal commands, keyboard and mouse input, browsing websites, web server activity, printing, running VMs and containers, changing connectivity settings (e.g., bluetooth, network, and VPN), and interacting with USB devices. With our templating, we demonstrate attacks on all systems, with a temporal resolution in the range of 0.2 ms to 11.5 ms.

On Linux, we demonstrate a local inter-keystroke timing attack with 93.1 % to 100 % F_1 score across 7 users, a remote (SSH) inter-keystroke timing attack with 100 % F_1 score, an end-to-end authentication UI redress attack on KDE Plasma 6 (Wayland), and a website fingerprinting attack on the top-100 websites with an F_1 score of 87.9 % in an open-world setting. On Windows, even worse, we show that any unprivileged user can leak the full paths of *all* files even without read permissions on a directory (e.g., users' home directories), directly leaking user and application file usage, exploitable, e.g., to reliably monitor website visits with an F_1 score of 97.8 % on the top-1000 websites. On macOS, we observe the least amount of leakage, which is still sufficient to monitor specific application starts and interactions. This shows that file-notification attacks are a generic security problem, cross-cutting across all modern operating systems despite their independent development.

CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

Side-Channel Attack, Operating System Security, File Notifications

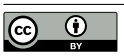
ACM Reference Format:
Sudheendra Raghav Neela, Xufan Zhao, Jeanette Angelika Wultsch, Hannes Weissteiner, Florian Draschbacher, Stefan Gast, and Daniel Gruss. 2026. File Notification Attacks: Templating and Exploiting Side-Channel Leakage from the File-Notification Systems on Linux, Windows, and macOS. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846521

1 Introduction

Many applications allow users to work with files even if they are modified in the file system simultaneously. For instance, a PDF viewer may need to monitor an opened PDF file to re-render the PDF when the user recompiles a LaTeX document. The application could simply busy-wait and poll the file for changes [29]. However, this strategy would be inefficient, wasting system resources and energy. Hence, modern operating systems like Linux, Windows, and macOS have built-in subsystems to monitor file events: `inotify` [38], `ReadDirectoryChangesW` [49], and `FSEvents` [3], respectively.

File monitoring subsystems allow arbitrary user processes with read access to a directory to monitor any file operation within that directory. For any file operation, e.g., access, write, open, and close, the monitoring subsystem raises a notification event to the subscribing (potentially malicious) process. Prior work monitored these notification events to infer application starts on Android [1, 67]. However, in the same threat model (second user, local code execution) an attacker can also exploit other side channels, e.g., the page cache side channel [6, 18, 57, 70] which can leak fine-grained information while still being hardware-agnostic. Similarly,

*Also with A-SIT Austria.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846521

page deduplication attacks can leak the presence of specific data in memory across users [17, 74] in a hardware-agnostic way. An attacker can also mount cache side-channel attacks [43, 60, 83] within the same threat model to leak significantly more information.

Another aspect to utilizing side-channel leakage in an attack is to determine the spatial and temporal attack points, e.g., addresses or offsets as well as the precise time when to mount an attack. Since the search space for an attacker can be very large, it can be difficult to discover the right leakage points in a manual analysis. Template attacks [8, 65] address this problem by automatically exploring the search space and map the cause-effect relation between secrets and side-channel leakage to create templates. Gruss et al. [20] mounted template attacks on CPU caches, allowing to partially monitor keyboard input of users. File monitoring subsystems also expose a vast search space, since any file in the system is a potential side-channel leakage point, that has not been explored yet. Consequently, it remains unclear whether file-notification information can be utilized in more powerful attacks across different operating systems.

In this paper, we present the first generic file-notification template attacks on Linux, Windows, and macOS, revealing leakage on various system and user activity. Our template attack consists of two phases: the templating phase, and the attack phase (see Figure 1). The attacker-controlled templating phase is a systematic and semi-automated scan discovering file-notification information leakage that an attacker can exploit, using a cause-effect analysis between user activities and file-operation events. We template a broad range of user activities including running terminal commands, physical keyboard and mouse input, browsing websites, web server activity, printing, running virtual machines and containers, changing connectivity settings (e.g., bluetooth, network, and VPN), and interacting with USB devices. While performing these user activities, we monitor all file-operation events occurring in the entire system's file system in a single pass. Consequently, in contrast to prior template attacks, the runtime of our templating phase is dominated by the user activity performed and not by the search space, as we do not mount a side-channel attack but obtain information directly from the operating systems as an oracle, effectively allowing monitoring all files on the system, even with a single pass of user activity. On Linux, `/dev` and `/usr` particularly expose abundant side-channel leakage, e.g., local and remote inter-keystroke timing with just 2 files in `/dev` and website fingerprinting with 1 448 files in `/usr`. The result is a template, mapping system, user, and application activities to specific events on files.

In the exploitation phase, we use these templates and the file-notification subsystems to spy on user activity. We find that the subsystems are very efficient with a maximum of 0.21 % CPU overhead, allowing us to monitor file-operation events with a temporal resolution in between 0.2 ms to 11.5 ms on Linux, Windows, and macOS. On Linux, we demonstrate local and remote (SSH) inter-keystroke timing attacks with F_1 scores between 93.1 % to 100 % (66 % of keys within 1 ms) across 7 users, a practical authentication UI redress attack on KDE Plasma 6 on Wayland, and a website fingerprinting attack on the top-100 websites with 87.9 % F_1 score in an open-world setting. We also investigate Android, which is based on Linux, using our templating methodology beyond prior work [67] to find a novel unprivileged primitive which reveals the existence of permission-protected files that apps cannot access, enabling an attacker to infer user activity, e.g., with WhatsApp. On Windows, we show that any unprivileged user can leak the full paths of *all* files even without read permissions on a directory (e.g., users' home directories), enabling direct leakage of user and application file usage such as website visits with 97.8 % F_1 score on the top-1000 websites with just one pass. On macOS, we observe the least amount of leakage but still can observe user, application, and system behavior, e.g., starting and interacting with applications.

The foundation of our attack is the fact that unprivileged users can utilize the file-notification subsystem. Even worse, while Windows and Linux have entirely independent implementations for file notifications, we discovered ways to obtain events for files **without read permissions** on both Windows and Linux. That is, an unprivileged attacker can monitor events on files that the attacker cannot even read, e.g., other users' personal directories on Windows. While seeming harmless, we show that the existence of files and filenames can already leak significant information. Furthermore, this is the foundation for our most powerful attacks on Windows and Linux, namely direct leakage (*i.e.*, not a side channel) of website visits in real-time as well as our inter-keystroke timing attacks. We responsibly disclosed this issue to the affected vendors including Microsoft, who stated in their response that this privacy-compromising undocumented behavior is present *by-design*.[1]

Our work shows that file-notification attacks are a generic security problem, cross-cutting across all modern operating systems despite their independent development. Therefore, we conclude that mitigations, which include improving fine-grained user permission models and enhancing file-notification APIs, are crucial to protect user privacy and the operating system's security.

In summary, we make the following main contributions:

(1) We present the first systematic and semi-automated approach to discover file-notification information leakage on Linux, Windows, and macOS using a template attack.
(2) We discover that both Linux and Windows allow unprivileged users to monitor events on files without read permissions, with significant privacy implications.
(3) We demonstrate that we can monitor files with a temporal resolution in the range of 0.5 ms to 11.5 ms on Linux, Windows, and macOS. On Linux, we can observe up to 250 000 events per second on our test systems.
(4) We evaluate our attacks on numerous targets and demonstrate local and remote inter-keystroke timing attacks with F_1 scores between 93.1 % to 100 % across 7 users, UI redress attacks on KDE Plasma 6, and a website fingerprinting attack on the top-100 websites in an open-world setting with an F_1 score of 87.9 % on Linux and an F_1 score of 97.8 % on the top-1000 websites on Windows.

Responsible Disclosure. We responsibly disclosed our findings (October 2025) to the security teams of Linux, Windows, macOS, and KDE. All vendors acknowledged our findings[1] and Linux has deployed partial patches. We mention further details in Appendix B.

Outline. Section 2 provides background on side channels and filesystem events. Section 3 presents our templating and attack

[1] We responsibly disclosed our findings to the Windows Security team. In their response, they stated that our findings were *by-design*. Furthermore, they state that it "only reveals file names and paths within another user's profile directory [and] does not expose file contents or sensitive data".

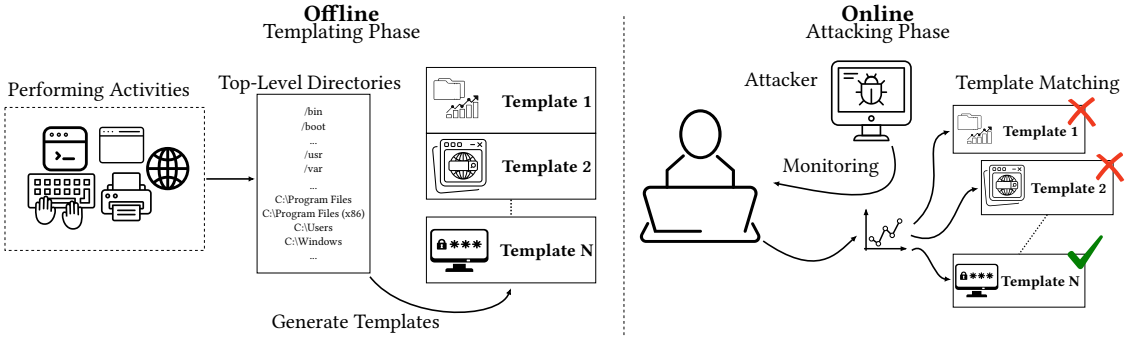


Figure 1: An overview of the two phases of file-notification templating attacks: Templating and Attacking.

methodology. Section 4, Section 5, and Section 6 evaluates attacks on Linux, Windows, and macOS. Section 7 discusses related work and mitigations. Section 8 concludes.

2 Background

In this section, we introduce background on side-channel attacks, OS side-channel attacks, and filesystem events & notification APIs.

2.1 Side-Channel Attacks

In a *side-channel attack*, the attacker infers secret information from a side-effect of processing that information. Such side-effects include energy [31], timing [30], and cache accesses [83] among many other channels. Secret information recovered from such attacks includes cryptographic keys [30, 31, 83], user interaction [20, 79], browsing histories [79] and internal operating system states [47]. In a *covert-channel* setting, a cooperating sender purposefully modulates a signal into such a side-effect to establish an illegitimate communication channel with a cooperating receiver. If different secret inputs produce differences in the side channel, template attacks are a powerful form of attack [5, 7, 8, 65, 69, 70], working in two steps: In the attacker-controlled, offline **templating phase**, they collect and measure cause-effect relations between secret input and side-channel output. In the **attack phase**, they use the template to infer secret inputs based on side-channel output.

Website Fingerprinting. A popular side-channel application is *website fingerprinting*: The attacker records a side-channel trace from a victim and matches it against a set of prerecorded and labeled traces. Different websites cause different network traffic patterns when transmitted [22, 66] and also different patterns of various local effects on the machine rendering the website [24, 27, 57, 79, 80], forming distinguishable fingerprints.

Inter-Keystroke Timing Attacks. Another line of research focuses on recovering user input, often from *inter-keystroke timings*, *i.e.*, the time intervals between key presses [23, 42, 73, 79, 85]. An attacker can exploit these timings to infer information about the user input, as they depend on the finger movements across the keyboard, and consequently, on the text being typed. Prior work has shown that these timing differences can be used to recover the typed text with machine learning and LLMs [64, 73, 85].

UI Redress Attacks. Instead of recovering a victim user's password from a side channel, an attacker can also draw a fake password prompt over a real one, intercepting input from the unsuspecting user. To lend credence to these *UI redress attacks*, the attacker has to detect the event that makes the real, legitimate prompt appear on the screen [15, 18, 28, 57, 68] via a side channel e.g., via page cache [18, 57], HMB [28], CPU utilization and network activity [10].

2.2 OS and Mass-Storage Side Channels

Besides the hardware, the OS itself can also leak sensitive information via side channels due to internal buffers and caches. A popular target is the page cache, which has been exploited in multiple prior works [6, 18, 57, 70]. Gruss et al. [18] presented a covert channel with up to 7 kB/s on Linux and 273 kB/s on Windows. Their inter-keystroke timing side channel achieves a temporal resolution of 149 ms. However, the `mincore` syscall has been patched [32] and it cannot be utilized for this side channel anymore. Neela et al. [57] recently presented primitives which interact with the page cache, introducing attack primitives, covert channels reaching 37.7 kB/s, and inter-keystroke timing with temporal resolution of 0.8 µs, which function on modern Linux systems. There are also side channels directly related to file operations, e.g., due to sync operations and write buffers [9, 25]. Several works attack the kernel itself through timing differences within kernel subsystems [33, 46, 72].

Even if the page cache cannot be attacked, an attacker can simply focus on the next layer of the memory hierarchy: the storage device. Trochatos et al. [78] and Giechaskiel et al. [16] presented the first SSD covert channels using attacker-controlled FPGAs. While their channels had capacities < 1 bit/s, they already demonstrated the potential of such attacks. Recent work on SSDs [27, 28, 81] has shown covert channels via contention and hardware-specific buffers.

2.3 File Notification APIs

File notification APIs enable userspace applications to subscribe to certain events on files or directory within a filesystem, such as files being opened, modified, or deleted. There are many use cases for such APIs, for instance, modifications of an opened file in an editor [53, 84]. File managers can subscribe to events for the currently viewed directory and immediately update the view when a change is detected [29]. File synchronization programs can

automatically start a synchronization to another machine whenever any of the files in the monitored directory changes [45]. Other use cases include logging and on-access virus scanning [11, 29].

In general, the application registers a list of the files or directories to monitor with the kernel, together with events, e.g., opening, reading, modification. Applications either wait using a blocking system call, or get notified via a callback function or a signal handler. Over time, multiple file notification APIs have been developed independently for Linux, Windows and macOS.

Linux. Linux has three kernel APIs for file notifications: `dnotify` [37], `inotify` [38] and `fanotify` [35]. The older `dnotify` API monitors entire directories, but has multiple shortcomings: It cannot monitor single files, cannot deliver details about the event to the application, requires an open file descriptor for every directory to monitor, and delivers signals to the application. Consequently, `dnotify` is mostly superseded by `inotify`, introduced in Kernel 2.6.13 [41]. The `inotify` API enables applications to monitor single files and delivers events via a readable file descriptor, including details such as the name of the relevant file and the exact event type [29, 38]. Additionally, `inotify` also provides notifications for opening and closing files, and removes the need for a file descriptor for every monitored file. Android's `FileObserver` class is also based on `inotify` [76]. The third and newest API is `fanotify`, introduced with Kernel 2.6.36 [35]. The `fanotify` API offers additional functionality, like monitoring an entire filesystem and performing additional permission checks in userspace, mostly relevant for antivirus software and userspace storage management [11, 35, 61]. Most of the extended functionality requires elevated privileges [36]. Both `fanotify` and `inotify` are implemented via the same kernel code paths, with the userspace interface being the major difference.

Windows. The `ReadDirectoryChangesW` winbase API function provides changes within a specified directory [49], the earliest system supporting this API being Windows 2000 [59]. The function requires an opened directory handle as an input argument. Other methods to receive file-operation notifications also exist, such as the dotnet FileSystemWatcher Class [52] or the file API's `FindFirstChangeNotificationA` [50]. However, we find the dotnet class to be a wrapper for `ReadDirectoryChangesW` and the file API's function to support fewer events than the winbase API's function (`Last_Access` and `Change_Creation` are not reported). As on Linux, `ReadDirectoryChangesW` is available to unprivileged users with read permissions on the directory to monitor.

macOS. On macOS, there are two file-notification subsystems: the file system events (`FSEvents`) API [3] and `kqueue`, the latter being technically part of the BSD tree in the XNU kernel [2]. Compared to `FSEvents`, `kqueue` supports fewer file-operation events overall. Furthermore, `kqueue` works on file descriptors that are opened by the current process, known to not scale well with the number of watched files [14]. The `FSEvents` API raises file-operations events by monitoring a directory. The API requires an array of readable directories for watching and is available to unprivileged users.

3 File-Notification Templating Attacks

In this section, we discuss the file-notification features of Linux, Windows, and macOS, and develop the first file-notification templating attacks on these systems. We first enumerate the file-notification events supported by the three systems, exploring the similarity and differences between events across the systems. Second, we present the cross-user threat model in which we mount our attacks. Finally, we describe a systematic technique to investigate side-channel leakage arising from file-notification events. We evaluate our file-notification attacks on Linux, Windows, and macOS systems in Section 4, Section 5, and Section 6, respectively.

3.1 File Events

The three operating systems implement their own subsystem to generate events upon file or directory operations. Across the three systems, the procedure for receiving notifications about such operations shares similar characteristics: (i) a process requests to subscribe to file-operation events; (ii) the subsystem checks the calling process for read permission for the file or directory; (iii) the subsystem creates an event queue from which the process can read incoming file-operation events; and (iv) the subsystem pushes file-operation events into this queue when they occur. In Table 1, we provide an overview of the events generated by the three subsystems. Primarily, some events are generated on one system but not in others. Certain events in Windows and macOS correspond to a single event in Linux, which results in multiple events corresponding to one Linux event. On Linux with `inotify` [29, 38], all events raise notifications for both files and directories, with the exception of `CLOSE_WRITE`, and `MODIFY` which raise notifications only for files [38]. The singular file-operation event generated on Windows but not on Linux or macOS is the `SECURITY` event [51], triggered by updates to the file's security metadata. On macOS, there are two unique file-operation events, `FinderInfoMod` and `Cloned`, related to macOS' file manager and file clone events, respectively. Interestingly, macOS also supports file-operation events for the mounting and unmounting of a volume: `Mount` and `Unmount`.

An important distinction between Linux and Windows is the behavior of Linux's `ACCESS` and Window's `LAST_ACCESS`, the event raised upon a file access: Linux generates an event upon every access while Windows' implementation does not. On Windows, a `LAST_ACCESS` event is generated when the "Last Access Time" file-parameter is updated. Since 2006, Windows has disabled updates to the Last Access Time file-parameter for performance reasons [75]. Therefore, we disregard the `LAST_ACCESS` file-operation event on Windows. Regardless, file-notification attacks are still possible on Windows and macOS despite the lack of a file-access event.

3.2 Threat Model

Similar to prior work [4, 18, 19, 21, 57, 70, 82, 83], we assume an unprivileged attacker who executes code as a separate user on a victim's system. This attacker who mounts the file-notification attack on Linux, Windows, and macOS requires one fundamental precondition to be met: read access to a target set of files. As user directories are read-protected from other users, this protection requires the attacker to monitor globally readable files. This limitation is shared with prior work [18, 19, 57, 70, 83] and restricts the number of files on which an attacker can mount an attack. On Linux machines with systemd, services often run as unprivileged users, separate users and can become compromised, e.g., XZ Utils.

Table 1: Equivalent File-Notification Filters

Description	Linux `inotify`	Windows `ReadDirectoryChangesW`	macOS `FSEvents`
File was accessed	`ACCESS`	`LAST_ACCESS*`	—
File attributes or extended attributes were modified	`ATTRIB`	`ATTRIBUTES`	`InodeMetaMod` `XattrMod` `ChangeOwner`
File that was opened write access was closed	`CLOSE_WRITE`	—	—
File that was not opened with write access was closed	`CLOSE_NOWRITE`	—	—
File was created	`CREATE`	`FILE_NAME` `DIR_NAME` `CREATION`	`Created`
File was deleted	`DELETE`	`FILE_NAME` `DIR_NAME`	`Removed`
Monitored file was deleted	`DELETE_SELF`	—	—
File was modified	`MODIFY`	`LAST_WRITE` `SIZE`	`Modified`
Monitored file was moved	`MOVE_SELF`	—	—
Monitored file was renamed	`MOVED_FROM`	`FILE_NAME`	`Renamed`
Monitored file was renamed	`MOVED_TO`	`FILE_NAME`	`Renamed`
File was opened	`OPEN`	—	—
File's security descriptor was changed	—	`SECURITY`	—
File was cloned	—	—	`Cloned`
File's Finder metadata was modified	—	—	`FinderInfoMod`
Volume was mounted	—	—	`Mount`
Volume was unmounted	—	—	`Unmount`

Linux `inotify` events are prefixed with `IN_`, Windows `ReadDirectoryChangesW` events with `FILE_NOTIFY_CHANGE_`, and macOS `FSEvents` events with `kFSEventStreamEventFlagItem`; *`LAST_ACCESS` on Windows is not perfectly equivalent to Linux's `ACCESS`.

However, the set of globally readable files is still vast, and we show in this paper that the operations on these files leak enough information to mount file-notification attacks. Importantly, employing the file-notification subsystems on all three operating systems does not require *any* special elevated privileges or capabilities, *i.e.*, the attacker runs unprivileged. These subsystems also often do not get updated, reducing complexity for the attacker across kernel versions. Furthermore, file-notification attacks are hardware-agnostic and, thus, an attacker does not require any special hardware.

3.3 File-Notification Templating

We present a systematic technique to template side-channel information leakage about user, application, and system using file-operation events on all three systems. Our approach is semi-automated, involving manual triggers and a programmatic method to profile the set of file-operation events that occur before, during, and after the trigger. If a trigger exhibits a high correlation with a file and its associated events, we consider this correlation as side-channel information leakage, *i.e.*, we template the leakage. For example, if launching an application results in a particular set of files being accessed, temporary or log files being created, and configuration files being modified, we then correlate this set of files and the events performed upon them to the application's launch. An attacker can determine attack targets such as applications or binaries using unprivileged process reporting commands (`ps`, `tasklist`), or monitoring file-operation events in directories where binaries or shared libraries typically reside, or use page-cache-based inference [57] to determine already-running binaries when kernel hardening features are active, e.g., `hidepid` on Linux.

Templating Phase. Our approach involves recursively watching all system-supported file-operation events occurring in a directory and its sub-directories while a manually triggering an event. In our evaluation on all three systems, we monitor the root directories for all file-operation events. However, it may not be possible to watch a large number of files as the system may either enforce limits or become unstable. For example, Linux's `inotify` subsystem explicitly defines the maximum number of watches at `/proc/sys/fs/inotify/max_user_watches`. In such cases, we either increase the number of watches (privileged), or split the monitoring across top-level directories over multiple runs. Windows and macOS do not enforce such a limit. Since the templating phase runs on the attacker's system, they are free to modify their system in any way.

Profiling Activities. For our evaluation on Linux, Windows, and macOS systems, we perform a general set of activities while recording file-operation events in the root directories. Not all activities produce file-operation events and, therefore, notifications. Only files with a positive cause-effect relation are included in the template and discussed further in this paper.

These activities include (i) terminal commands, (ii) physical keyboard and mouse input, (iii) installing, launching, and uninstalling

Table 2: Experiment Machines For Linux

	Processor	Distribution	Kernel
$\mathcal{M}_1$	AMD Ryzen 7 PRO 5850U	Ubuntu 22.04.5	6.8.0
$\mathcal{M}_2$	AMD Ryzen 7 5800X3D	NixOS 25.11	6.17.7
$\mathcal{M}_3$	Intel Core i7-1260P	Ubuntu 24.04.3	6.8.0
$\mathcal{M}_4$	Intel Core i7-10510U	Ubuntu 24.04.1	6.14.0
$\mathcal{M}_5$	Intel Core Ultra 7 155U	Debian 13	6.12.48
$\mathcal{M}_6$	AMD Ryzen 7 8845HS	Gentoo	6.6.100
$\mathcal{M}_7$	AMD EPYC 8024P	Ubuntu 22.04.5	6.11.0

applications, (iv) interacting with applications, (v) browsing websites, (vi) web server activity, (vii) printing files, (viii) running virtual machines, (ix) running docker containers, (x) changing bluetooth, network, and VPN settings, and (xi) interacting with USB devices.

For our evaluation, we choose popular applications for activities (iii) and (iv): Steam (video game client), Zoom (video conferencing software), and Discord (internet communication). Depending on the idiosyncrasies of each subsystem, we evaluate additional targeted activities based on the type of information leakage. Tests are run on personal devices under typical background activity and performed manually, as some activities are not easily automatable. Each is repeated multiple times, including after a reboot, to confirm consistency. If no notification appears, we wait 20 s to 30 s before retrying to confirm the absence of a cause-effect relation.

Attack Phase. In the attack phase, our attacker program subscribes to the relevant events identified in the templating phase. Typically, only a few files are relevant for a specific user action. Consequently, the attack phase does not require the maximum number of watches to be increased. Depending on the type of attack, the number of watches can range from a singular file (e.g., detecting whether a binary is executed), to watching a directory (e.g., temporary files created or deleted), or a set of files and directories (e.g., access patterns of shared libraries and configuration files).

4 File-Notification Attacks on Linux

In this section, we mount file-notification attacks on Linux. We first describe our experimental setup, providing details regarding hardware-agnosticism and specifics of the activities we execute. Second, we perform our activities and discuss observations while monitoring all the root-level directories. Afterwards, we evaluate and mount attacks using the insights from our observations. Finally, we investigate Android as a Linux case study, extending prior work [67] to find a novel unprivileged primitive which reveals the existence of permission-protected files that apps cannot access.

4.1 Experimental Setup

Experiments were performed across multiple machines (summarized in Table 2), primarily on $\mathcal{M}_1$ with 32 GB of RAM running Xorg and KDE Plasma 5 (5.24.7). As our attacks are hardware-agnostic, we validate them on the other machines as well.

We mount our investigations and attacks in the cross-user threat model using `inotify`, creating an unprivileged user and monitoring all activity as this newly created user. A challenge on Linux distributions is non-conformity to the Filesystem Hierarchy Standard (FHS) [44]. Some distributions or package managers use non-standard file paths, e.g., `/snap` and `/nix`. Furthermore, despite being FHS compliant, a Linux distribution and its package manager may place files or use symbolic links in distro-specific directories. For our investigations, we used Ubuntu systems after version 22.04.

4.2 Templating Phase on Linux

We employ our templating approach (see Section 3.3) to monitor all root directories. To avoid being limited by the maximum number of watches, we increase the limit for the templating phase. As the learned templates can be transferred between systems, we do not need the increased watch limit during the attack phase. We describe the key observations from our investigation.

4.2.1 The Unreadable File Bypass. Despite not having read-access to a file, users can still receive file-operation notifications if the file resides in a readable directory.For example, unprivileged users cannot read the system log located at `/var/log/syslog`; thus they cannot use `inotify` to receive file-operation notifications on the file directly. However, the parent directory, `/var/log`, is publicly readable. Therefore, unprivileged users can use `inotify` to watch `/var/log`, receiving file notifications for all files *within* the directory, including `syslog`. Applying this principle to other directories, such as `/dev`, can enable significant information leakage.

4.2.2 Access to Device Files. The device files in `/dev` represent devices on the system [58]. On our system, `/dev` contains 278 files (excluding symbolic links) that are non-readable, yet can trigger file-operation notifications. Observing usage patterns of directories in `/dev` leaks significant information about user, system, and application behavior. For example, Electron apps (Discord) momentarily create and destroy files in `/dev/shm` depending on activities within the application. Other directories leak information specific to devices, e.g., `/dev/snd/` when a user interacts with sound settings, `/dev/bus/usb` when a USB is plugged in, or `/dev/tun` when a VPN is active and packets are being sent over the tunnel. Especially notable is leakage from `/dev/input`, which triggers notifications on input from hardware such as a keyboard and mouse. In Section 4.4.1 we exploit this behavior to mount an inter-keystroke timing attack. We also detect when text on shells (pts [39] or tty [40]) are updated, including over SSH connections. We mount an inter-keystroke timing attack on a remote server in Section 4.4.2. The Linux kernel notifies the attacker which exact file was accessed when watching a directory. Each input device in `/dev/input/` maps to a file, e.g., `event0` is the keyboard, `event6` is the touchpad. However, since an attacker will not know which device corresponds to a file, the attacker can use access times to reveal the device type and associated files: rapid access bursts indicate touchpad usage while sporadic accesses indicate keyboard presses.

4.2.3 Detecting Binary Execution. The `/usr/bin` and `/bin` directories contain system-wide executable binaries, which are installed during system installation or by a privileged user (`/bin` is symbolically linked to `/usr/bin`). By subscribing to file-*access* notifications for files within these directories, a user can infer when a binary is executed, which can be problematic for security-relevant binaries such as PolicyKit's `pkexec` [63]. In Section 4.4.3, we mount

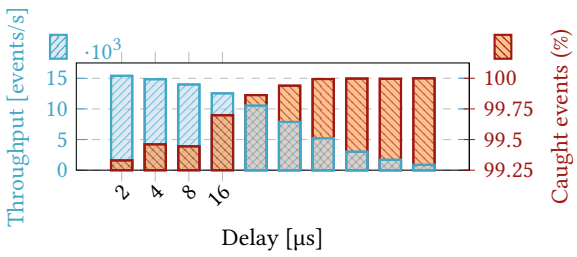


Figure 2: Single-file `inotify` notification throughput with 16 384 accesses tested across varying access delays. Shorter delays result in more missed notifications, with the percentage increasing significantly at fewer accesses per iteration.

an authentication UI redress attack on KDE Plasma 6 (Wayland) when detecting accesses to `pkexec`. Similarly, `/sbin` contains system binaries which can only be executed by a privileged user, and accesses to files within this directory leak administrator activity.

4.2.4 Linux `sysfs`. The `/sys` directory is a Linux-specific virtual filesystem (`sysfs`) that exposes information about hardware, drivers, configurations and other options to user space, with a varying structure based on the kernel version [44, 55]. We find that user or system requests to hardware and network settings, e.g., WiFi and Bluetooth, raise specific file-operation events in `/sys/devices`. In docker containers, while `inotify` is disallowed [13, 54], we unexpectedly find that some paths within `/sys` trigger events *between* containers of different images. We discuss specifics in Appendix C.

4.2.5 Application Leakage. Other top-level directories leak application-related information. Many applications access `/etc` and `/opt` during startup, as `/etc` contains configuration files for applications and `/opt` contains other application-specific files. Loading shared libraries from `/lib` (`/lib`, `/lib32`, `/lib64`, `/libx32`) also generates notifications, revealing usage patterns. The `/var` contains files with variable data such as logs, system cache, print jobs, or web server files. Monitoring `/var` reveals system-level information, e.g., package manager updates, authentication attempts (`auth.log`, `syslog`), or printing information (`cups`). Accesses to webpages hosted by Caddy HTTP server generate access notifications to the served file. We find that `/usr` leaks the most information about user and application behavior, as most user interaction with applications causes file operations in `/usr/share`. On KDE Plasma 5 (Xorg), every key press in a search menu generates file-operations events in `/usr/share/kservices5/searchproviders/`, similarly on GNOME in `/usr/share/icons/hicolor/index.theme`. In Firefox, browsing different websites results in accesses to different font files in `/usr/share/fonts/`, which we demonstrate can be used for website fingerprinting in Section 4.4.4.

4.2.6 Temporal Resolution. We measure `inotify` temporal resolution with a file reader (`read` syscall) and a file monitor, both recording timestamps for their operations, with `sysstat` reporting only a 0.2 % CPU overhead. We find that resolution stays below 10 µs across all systems. $\mathcal{M}_1$ averages 10.0 µs (n=1000, σ=3.9 µs), $\mathcal{M}_3$ at 6.4 µs (n=1000, σ=2.6 µs), and $\mathcal{M}_2$ at 5.9 µs (n=1000, σ=1.9 µs). Under complete CPU stress, resolution on $\mathcal{M}_1$ degrades to 14.71 µs on average and up to 50.82 µs under maximal `inotify` stress.

4.3 Notification Throughput & Covert Channels

To evaluate the performance of `inotify`-based attacks, we first measure the maximum file notification throughput *i.e.*, how many notifications can be received per unit time (measured in events/s), representing the maximum information that can be leaked. We evaluate two scenarios: single- and multi-file notification throughput.

4.3.1 Experimental Setup. All throughput experiments were conducted on $\mathcal{M}_3$ with 48 GB of RAM. To minimize unexpected file accesses, we created communication files in `/tmp`. No additional isolation measures were taken, and the machine was used for regular tasks, showing that `inotify` is resilient to general system noise.

4.3.2 Single-File Notification Throughput. In this scenario, the sender first waits until the receiver has set up a watch on a file. Then, the sender triggers access notifications by repeatedly reading from the file with a variable delay between each read access for a configurable number of iterations. Concurrently, the receiver measures the time between notifications. Afterwards, the sender signals the receiver to stop waiting for notifications.

Results. We perform the single-file experiment with delays of 2 µs to 1 024 µs between accesses, triggering 2 to 16 384 notifications per experiment. With delays of 64 µs or less, we observe lost notifications during each experiment. With a delay of up to 512 µs, notifications are still lost during some experiment runs. However, with delays of 1 024 µs or more, we reliably receive all notifications on each of our attempts. Notably, even with a large number of file accesses, we only lose a small number of notifications per run, even with very short delays. Surprisingly, with fewer iterations, we lose the same number of notifications, resulting in higher relative loss. Figure 2 shows the throughput and percentage of lost notifications with 16 384 iterations, averaged over 10 runs.

Experiments on different machines shows that this loss of notification events depends on the system performance. On $\mathcal{M}_2$, a delay of 32 µs is sufficient to receive almost all notifications, in contrast to the 512 µs required on $\mathcal{M}_3$. We perform further experiments using $\mathcal{M}_3$, as it provides a lower bound on achievable throughput.

4.3.3 Multi-File Notification Throughput. To increase throughput and reliability, we can transmit information using multiple files. Each access now contains $log_2(n)$ bits of information, where n is the number of files used. Thus, by using 256 files, we can transmit $log_2(256) = 8$ bit per access. We evaluate two variants: using read operations and using open-close operations. On our systems, processes have a default limit of 1 024 open file descriptors, limiting the amount of information that can be transmitted in the read variant. On $\mathcal{M}_3$, each user can create up to 65 536 `inotify` watches, allowing us to transmit almost 2 bytes per notification using open-close operations (other programs also use watches, limiting the maximum to 64 000.). On other systems, `inotify` limits are much higher, up to 524 288 watches, allowing for even more bits per notification.

Setup. The multi-file experiment is similar to the single-file version. However, instead of using a single file for communication, the sender creates the requested number of files (between 32 and 64 000). The receiver sets up `inotify` watches on all files. To send data, the

Table 3: Multi-File Notification Throughput

Files	Read	Open-close
32	78.93 kbit/s	74.09 kbit/s
64	94.97 kbit/s	90.43 kbit/s
128	110.44 kbit/s	111.96 kbit/s
512	140.31 kbit/s	134.42 kbit/s
1 000	155.91 kbit/s	150.12 kbit/s
1 024	N/A	152.22 kbit/s
8 192	N/A	198.80 kbit/s
16 384	N/A	219.32 kbit/s
32 768	N/A	230.39 kbit/s
64 000	N/A	254.80 kbit/s

sender either reads 1 byte from one of the files (read variant) or opens and closes one of the files (open-close variant), depending on the transmitted data. The receiver records the time between each received notification and the file which triggered the notification, to reconstruct the sent data.

Multi-File Results. We perform the experiment with varying delays, ranging between 0 µs and 256 µs, triggering between 2 and 262 144 notifications per experiment, and using between 32 and 64 000 files. In the multi-file setting, we reliably receive all notifications (even with no delay between accesses), and the order between notifications is preserved. Thus, the limiting factor for throughput in a multi-file setting is the time it takes to generate and receive notifications. In our measurements, when using the same parameters, the read variant achieves a 6 % higher throughput on average compared to the open-close variant, which can be explained by the additional overhead of a second syscall in the open-close variant. However, the open-close variant can transmit more bits of information per notification due to the higher number of files, allowing for a higher overall throughput. When using 1 000 files, with increasing iterations, the read variant achieves an average throughput of 161.02 kbit/s, while the open-close variant achieves 151.25 kbit/s on average. However, when using 64 000 files, which is only supported by the open-close variant due to file descriptor limits, we achieve a throughput of 250.07 kbit/s on average. We show the results for 65 536 iterations in Table 3. On systems supporting even more watches, the throughput can be increased due to the higher number of bits transmitted per notification, up to an extrapolated theoretical maximum of 297.56 kbit/s when using 524 288 files.

4.4 Evaluating Attacks

In this section, we evaluate file-notification attacks on Linux. First, we mount an inter-keystroke timing on `/dev/input`. Second, we demonstrate a remote inter-keystroke timing attack over an SSH connection. Third, we mount an end-to-end authentication UI redress attack on KDE Plasma 6 (Wayland). Finally, we demonstrate a website fingerprinting attack on the top-100 websites in an open-world setting, where we achieve an F_1 score of 87.9 %.

4.4.1 Inter-Keystroke Timing Using Device Files. As discussed in Section 4.2.2, unprivileged users can receive notifications from

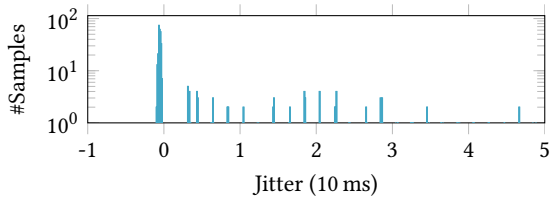


Figure 3: The jitter of the attacker-observed notifications on `/dev/input/event0` from ground truth (key press). 71.6 % of observations are within 4.5 ms and 66.6 % are within 1 ms of ground truth.

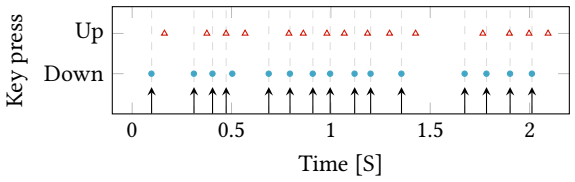


Figure 4: A two second sample of `inotify` notifications on `/dev/input/event0` while human typing. For each key press, we generally observe two access notifications, separated logically into key down and up.

files within `/dev/input/` due to the unreadable file bypass (see Section 4.2.1). This directory contains event files linked to different input devices [62], which trigger access and modify events on the corresponding input-event file when there is a physical input. On $\mathcal{M}_5$ using KDE Plasma 6 (Wayland), we see 16 event files (0-15) which correspond to various input devices including the keyboard, touchpad, power and sleep button, microphone, and lid switch.

For every key press, the attacker receives two access notifications: one for key down (press) and the other for key up (release). Crucially, we do not know which key was pressed and released, but only that a key press and release occurred. As these two events occur in pairs, we know that the first event is the key press. The timing *between* these key presses leaks information, leading to an inter-keystroke timing attack [56, 64, 73, 85]. Some typists and keyboards tend to generate multiple notifications (up to 8) for a single key down or up. However, in our experiments, these repeated notifications are within 5 ms of each other, and a grouping of this size filters out the repeated notifications to a single notification, similar as in cache side-channel attacks on keyboard input [20], without reducing the temporal accuracy for events that are further apart.

Interestingly, we find that the typing speed does not affect keystroke detection as much as the typing *style*. Every typed character always raises a key down notification, but not every release generates a key up notification. With an irregular rapid burst of typing, we observe the kernel occasionally drops or merges the key up notification, similar to Section 4.3.2. Instead of the key up notification, we observe a notification that aligns with the next key down. This behavior may be due to typists who overlap their key presses [12].

We evaluate our attack with two sets of human typists: 6 who typed an unseen prose (typing speed: 2.9 to 6.25 keys/s), and 1 pre-trained typist who practiced the prose 5 times (typing speed:

Table 4: Keystroke Detection Results of Untrained Typists

Typing Speed (keys/s)	2.9	3.3	4.4	4.9	6.15	6.25
F_1 Score	100 %	97.9 %	93.1 %	99.1 %	95 %	95.4 %
Temporal Resolution (ms)	0.05	0.09	0.75	3.53	3.33	2.21

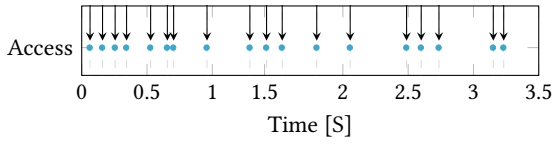


Figure 5: Over an SSH connection, a user's pseudo-terminal generates an access event for key presses. At a typing speed of 7.21 keys/s (arrows are key presses), an attacker receives access events within 2.9 ms from the ground truth.

7.14 key/s). Typists used their own keyboards on the same machine, $\mathcal{M}_5$. We record ground truth timestamps for each key press. In the attacker process, we monitor `/dev/input` and filter for accesses.

We determine the F_1 score calculation by manually matching `inotify` events to ground-truth keypresses: we cluster pairs of key-down (always first notification) and key-up (follows key-down notification) within 400 ms and then analyze the traces. We count a true positive for each ground-truth key-down and key-up event that matched an observed event, a false negative for each ground-truth event without a match, and a false positive for each observed event that could not be attributed to any ground-truth event. All classifications were made manually. We summarize the results of the 6 untrained typists in Table 4. The F_1 score is more than 90 % across all 6 users, ranging from 93.1 % to 100 %, and notifications arrive between 0.05 ms to 3.5 ms on average after the key press.

The F_1 score of the pre-trained typist is 100 %. We attribute the high F_1 score to the more regular typing cadence due to prior practice, combined with fewer mistakes. Figure 3 shows the jitter of the notifications from ground truth: 71.6 % of events are within 4.5 ms of ground truth, and 66.6 % are within 1 ms. On average, the attacker receives the notification 9.2 ms (n=402, $\sigma_{\bar{x}}$=0.8 ms) from ground truth. We show a short excerpt of the typing recording in Figure 4. Counter-intuitively, the attacker oftentimes receives a notification *before* the userspace ground-truth process records the key press. This bizarre behavior is due to `inotify`'s plumbing in the kernel, where the `fsnotify` function [34] can be called before the main action occurs (`inotify` shares code with `fsnotify` in the kernel). Overall, we show that the notifications generated on `/dev/input` can act as a reliable and precise keystroke detection.

4.4.2 Remote Inter-Keystroke Timing Over SSH. The previous attack leaked keystrokes on a *local* system. In this section, we demonstrate the same across users *remotely* over an SSH connection, where updates to a terminal generate *modify* events in the pseudo-terminal device file [39], `/dev/pts`, whenever the pts is updated.

On remote systems, we observe an additional access event upon a remote user's key press, and therefore, we filter only for access events on the pts, the exact pts number determinable by the `who` command. We observe only a single access event per key press, unlike the local inter-keystroke timing attack where two access events are generated. Similar to the local attack, an attacker does not know the exact key pressed, but rather only that a key was pressed. A crucial limitation in this scenario is that access notifications are only raised for key presses which cause textual updates. As such, key presses in remote input buffers which suppress textual feedback, e.g., the `sudo` password prompt, cannot be captured. Therefore, attackers cannot observe access notifications for such inputs.

We evaluate our remote inter-keystroke timing attack with two machines. On a local machine, a human typist typed 402 keys into the logged-in SSH session at an average of 7.21 keys/s. We use a keylogger on the local machine to record the ground truth. On the remote server ($\mathcal{M}_7$), located within the same network, we spawn an unprivileged process recording access notifications to `/dev/pts`. In Figure 5, we show a 3.5 s snippet of typing input and observed notification. We detect every key press without any false positives and negatives, thus achieving an F_1 score of 100 %. On average, our observations are 2.9 ms (n=402, $\sigma_{\bar{x}}$=0.02 ms) off the ground truth.

4.4.3 Authentication UI Redress Attack. We mount an end-to-end authentication UI redress attack [15, 18, 28, 57, 68], where an attacker's goal is to draw a fake authentication window (GUI password prompt) over a real authentication window on an unsuspecting victim user's system. We mount our attack on the latest KDE Plasma 6 (version 6.3.6) using the secure Wayland protocol on $\mathcal{M}_5$.

Instead of mounting our attack in the cross-user threat model, we assume a malicious or compromised process running as the victim user. Modern desktop environments implement focus-stealing prevention techniques which prevents unwanted windows from activating over focused windows [26]. In KDE Plasma 5 (Xorg) and Plasma 6 (Wayland), we find that focus-stealing prevention is not properly implemented for password dialogs and that any window can be drawn over password prompts. In contrast to X11, the Wayland protocol is designed to be secure, preventing any processes from reading global inputs unlike Xorg [48]. Therefore, in combination with properly implemented focus-stealing prevention, Wayland should prevent any process from stealing inputs.

The attacker uses `inotify` to determine when PolicyKit's `pkexec` [63] is executed by monitoring the binary for access events (see Section 4.2.3). Since `pkexec` (wired to `/usr/libexec/polkit-agent-helper-1`) is globally readable, an attacker can use `inotify` to monitor it directly. An access notification reliably indicates a system-password prompt is on screen, e.g., triggered by KDE Discover during updates, regardless of which application invoked `pkexec`. After `pkexec` is launched, the attacker launches a fake password prompt window covering the real prompt, misleading the victim user to enter their password into the fake prompt.

In our evaluation, the attacking program receives an access notification 167.9 ms (n=1000, σ=54.4 ms) on average after it was executed. From `pkexec` execution, the total time required to finish drawing our fake window is 374 ms (n=1000, σ=14.89 ms). On a 30 FPS display, this is 11.2 frames after `pkexec` was executed and does not account for time taken to draw the real password prompt.

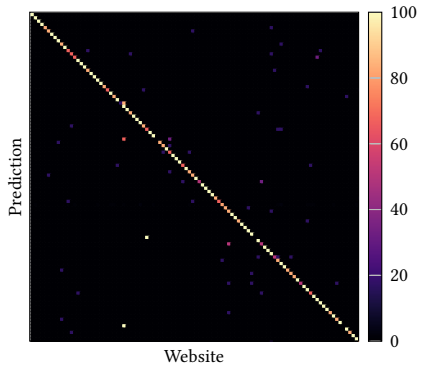


Figure 6: The classification heat map of our Linux website fingerprinting attack. The F_1 score is 87.9 %.

4.4.4 Website Fingerprinting. In this section, we demonstrate that an attacker can use `inotify` to mount a website fingerprinting attack on the top-100 websites in an open-world setting. Specifically, we find that visiting websites on Firefox results in different access patterns to system-wide fonts located in `/usr/share/fonts/`. An attacker can use these access patterns as a side channel to determine the website being visited. For example, we find `target.com` to access the most font files in the top-100 websites, averaging at 657 font file accesses. Other sites access fewer font files, such as discord, yahoo, or samsung, which access between 24-26 font files per visit.

We evaluate our attack on $\mathcal{M}_4$, and we use Firefox apt version 143.0. On $\mathcal{M}_4$, there are 1 448 font files present in the fonts directory. We downloaded the top-100 websites from dataforseo in September 2025, replacing unreachable websites with the next highest entries on the list. We open each website for 10 s in a random order to prevent any biases or artifacts from prior visited websites, totally 30 traces per website. We additionally downloaded 750 websites for the open-world category, each visited only once. With our collected traces, we train a K-Nearest Neighbors classifier with 20 % traces reserved for testing the classifier. With K set to 1, we achieve an F_1 score of 87.9 % in this open-world setting. We show the heat map of this classification in Section 4.4.4. We verified that our attack also works on the snap-installed versions of Firefox and Chrome.

4.5 File-Notification Attacks on Android

This section describes our results on Android, which differ from those on Linux. Prior work [1, 67] on Android was limited to detecting application startups using `FileObserver`, the Android wrapper around `inotify`. We go beyond prior work by identifying an unreadable-file bypass different from Linux, which arises from the interactions of different components of Android's scoped storage.

4.5.1 Threat Model and Experimental Setup. Our threat model differs from that of normal Linux due to the platform's diverging security model. On Android, there is no public file system accessible to unprivileged apps. Unprivileged apps can write to shared folders under `/sdcard` (e.g., `/sdcard/Downloads`), but their read access is limited. Listing any of these folders only yields subfolders and those files that have been created by the app itself. Android creates this per-app view through a FUSE layer that looks up file ownership in the system's `MediaProvider` database [77]. In our threat model, we assume an unprivileged malicious application without any permissions as in Android, each app is run as a different Linux user. We perform evaluations on a Google Pixel 9a and a Samsung Galaxy A54, both running the most recent Android 16.

4.5.2 Templating Phase on Android. We use `FileObserver` for templating with slight adaptations, limiting profiling to common smartphone tasks and excluding terminal commands, web server activity, and VMs & docker containers. Notably, while Android's per-app view restricts unprivileged apps to listing only self-created files under `/sdcard`, `FileObserver` bypasses this by reporting events on *any* file when watchers are mounted on `/sdcard` or its subfolders, a side-effect of Android's layered file system support.

Although prior work [67] identified a small portion of this behavior, they identified that `FileObserver` can monitor files in apps' private folders only in the *portable* SD card configuration, whereas we show this also applies to the *internal* (emulated) configuration. Furthermore, while they demonstrate that the existence of an installed app can be inferred by probing *folder* existence in `/sdcard/Android/data`, they do not identify that `FileObserver` can reveal the existence of concrete files within those folders, a stronger primitive that enables attacks such as inferring a user's WhatsApp activity. Most critically, they do not observe that the per-app view on shared external storage, which on unprivileged apps is restricted to files the app itself created, can be bypassed entirely via `FileObserver`, allowing an attacker to learn whether the user has downloaded files, taken pictures, or captured screenshots. Notably, `FileObserver` exposes *exact* filenames and paths even when a directory's contents cannot otherwise be listed.

Due to restricted file system access, Android templating is limited. On both devices, system file access leaks printing, first-time VPN setup, Wi-Fi activation, and USB device events. The Galaxy A54 additionally exposes `/dev`, leaking Bluetooth and cellular activation, and VPN traffic statistics via `/dev/tun`. Prior work [1, 67] demonstrated templating applications based on their startup behavior. To measure `FileObserver`'s temporal resolution, we implement two unprivileged Java apps where one writes to `/sdcard/Downloads` and the other monitors it, both recording timestamps around file system operations and events. The temporal resolution is slower than native Linux, averaging 0.36 ms (n=1000, σ=1.5 ms) on the Pixel 9a and 0.22 ms (n=1000, σ=0.1 ms) on the Galaxy A54.

4.5.3 Revealing Private Communication. `/sdcard/Android` holds per-app private data, protected from other apps by the OS's FUSE layer. However, the existence of these files, which sometimes is privacy-relevant information, still leaks through `FileObserver` events. We mount an attack on WhatsApp's private media and documents storage. Since WhatsApp auto-downloads received media and documents to its private folder, an attacker learns the exact time files arrive and can extract metadata from filenames, e.g., screenshots commonly include the captured app and timestamp in their file name. Moreover, the attacker is notified when such downloaded are deleted, revealing possible attempts to conceal communication.

5 File-Notification Attacks on Windows

In this section, we present the results of our file-notification template attacks on Windows using `ReadDirectoryChangesW` [49].

5.1 Experimental Setup

We perform all evaluations on an Intel i5-13600K running Windows 11 24H2 with 64 GiB RAM. To mount cross-user attacks, we create an unprivileged, non-Microsoft-account user, and monitor activity using this new user. The victim user is a Microsoft-account user.

5.2 Templating Phase on Windows

We set our monitoring program to watch all the top-level directories in `C:\`, including Program Files, Program Files (x86), ProgramData, Users, and Windows. We first observe the *full* path leakage of unreadable files when mounting a watch on `C:\` itself.

Leaking Full Paths of Unreadable Files. Watching `C:\` on Windows leaks events on *all* files regardless of read permissions, an undocumented behavior by design.[1] Notable are the leaked paths of files in users' home directories when files are created, modified, or renamed, as unprivileged users receive notifications with the complete file path. Since applications modify specific files in a user's `AppData` directory, an attacker can precisely track application usage and user behavior. In Section 5.3, we monitor cross-user website visits with an F_1 score of 97.8 % on the top-1000 websites.

User, Application, and System Behavior. Due to the lack of an access notification (see Section 3.1), our cross-user monitoring program receives much fewer notifications overall. Still, we are able to identify system, user, and application events which trigger modifications to system-wide files, *i.e.*, side-channel leakage still exists. For example, modifications to files in `C:\appcompat` reveal information about application compatibility, file activity in `C:\Prefetch` leak recently executed commands or executables, and `C:\System32` often has file modifications upon driver activity. In Section 5.4, we template events as described in Section 3.3.

Temporal Resolution. We measure temporal resolution with a file writer and an event monitor, both recording time, `TotalProcessorTime` reporting 0.21 % CPU overhead. Across 1 000 attempts, the temporal resolution is 0.64 ms (n=1000, σ=0.99 ms) on average.

5.3 Revealing Website Visits

In this section, we demonstrate how an unprivileged user can reliably monitor cross-user website visits in real time. Web browsers create or modify files for websites when visited, storing metadata and client-side data such as local storage, IndexedDB, cookies, or cached resources. In Windows, these files are stored in the `AppData` directory of users' home directories, in a path which contains the *website's URL*. Although files in users' home directories are private across users, using the unreadable-file bypass from Section 5.2 allows an unprivileged user to leak file paths across users.

By mounting a watch on `C:\`, we find that Firefox interacts with a website's directory when the website uses one of local storage, IndexedDB, or client-side cache, while Chromium-based browsers like Edge do so only for IndexedDB. On first visit, both browsers create a directory for the website, provided the site requires specific storage. On revisits, existing files in the same directory are modified, also raising notifications. In Table 5, we show sample paths.

We evaluate this cross-user website monitoring by visiting the top-1000 websites on both browsers (using Selenium), the website list downloaded from dataforseo.com. Simultaneously, an unprivileged cross-user program records changes in the browser-specific directories, noting all websites that were visited. Afterwards, we compare the visit order and recording to determine the F_1 score. While recording, we found 25 of the top-1 000 websites did not respond, and therefore we only consider the 975 responsive websites. While we performed a top-100 website fingerprinting attack on Linux (see Section 4.4.4), we recorded a total of 3 000 traces, as our classifier required 30 traces per site for training; on Windows, we recorded only 2 000 traces for the top-1000 websites, since our approach required 2 recordings per site (initial and confirmation).

Table 5 shows that Firefox creates directories for 95.7 % of websites, versus only 32 % on Edge. The F_1 score on Firefox and Edge is 97.8 % and 48.5 % respectively as there are no false positives. As previously noted, Edge creates directories for websites using IndexedDB while Firefox does so for any of the three storage mechanisms. Fewer websites use IndexedDB, leading to Edge's lower scores.

Unlike side-channel website fingerprinting approaches (e.g., Section 4.4.4), using `ReadDirectoryChangesW` produces no false positives. Since it acts as a direct oracle rather than measuring a side-channel effect, it reports exact website directory paths by design (see Footnote 1). Thus, by watching `C:\`, unprivileged users can reliably monitor cross-user website accesses with no false positives.

5.4 User, Application, and System Behavior

In this section, we detail our observations templating user, application, and system behavior (described in Section 3.3). We observe reproducible sequences of filesystem events that can be used to template application behavior and specific user interactions.

Application Behavior. Application installations, updates, and uninstallations generate distinct file-system modification patterns. User interaction with applications also exhibit patterns which can be templated. We take the examples of Steam and Zoom. When Steam starts, it creates and deletes the `.writable` file to its installed directory. Afterwards, it updates configuration files and initializes shader caches for graphics APIs such as OpenGL and Vulkan (in `C:\System32`). We also observe activities such as cache management for built-in browser components and certificate refreshing. When closing Steam, the application finalizes user and session data, shuts down web components, and flushes application logs.

On startup, Zoom initializes `WebView2` caches, updates its main database (`zoomus.zmdb.kvs.enc.db`), and modifies graphics driver logs. Entering a waiting room triggers repeated database journal updates, while joining a live session creates a dedicated meeting database, activates new `WebView2` caches, and updates hardware logs. Leaving removes session databases, updates the main database, and compresses diagnostics logs. On shutdown, Zoom cleans up lock, temporary, and GPU-related files.

Settings & Antivirus Interaction. While navigating Windows Settings, we observe corresponding filesystem activity. For example, updating Windows Update settings results in database activity present in `ProgramData`, and wallpaper changes result in updates to files in `TranscodedWallpaper`. Similarly, account management

Table 5: Percentage of Top 1 000 Websites For Which Directories are Created by Browsers

Browser	IndexedDB	Local Storage	Cache	Websites Leaked (F_1 %)	Path (Visiting **target.com**)
Firefox	●	●	●	97.8 %	`Roaming\Mozilla\Firefox\Profiles\XXXXXXXX.default-release\storage\default\`**`https+++www.target.com`**`\`
Edge (Chromium)	●	○	○	48.5 %	`Local\Microsoft\Edge\User Data\Default\IndexedDB\`**`\https_www.target.com_`**`0.indexeddb.leveldb`

Both paths are prefixed with `C:\Users\USERNAME\AppData\`. Symbols denote whether the browser generates (●) or does not generate (○) directories with website names for the client-side storage type (column).

triggers updates in the identity cache located in `AppData`. Print-related jobs create, modify, and remove files in `System32\spool`.

Windows Defender's full-system scans inadvertently leak the paths of all files system-wide, including other users' files generated by past activity (e.g., browser files, documents, videos), by raising modified notifications on every scanned file. However, one limitation is the caching of results, *i.e.*, Defender appears to not rescan files if they are unmodified. Still, this full-path leakage occurs on the first scan, or if methods to clear Defender's cache are discovered.

Application Prefetch Patterns. Monitoring `C:\Windows\Prefetch` reveals application launches via prefetch files (`*.pf`), whose names embed the application or command name [71]. For example, the prefetch file for the SSH command is `SSH.EXE-<number>.pf`, Steam executable is `STEAMSYSINFO.EXE-<number>.pf`, and MS Edge is `MSEDGE.EXE-<number>.pf`.

6 File-Notification Attacks on MacOS

In this section, we mount file-notification template attacks using `FSEvents` on macOS.

6.1 Experimental Setup

We perform all investigations and evaluations on an 8-core Apple M2 CPU running macOS Sonoma (14.2.1) with 8GB of RAM. Similar as before, we create an unprivileged, non-AppleID user to monitor all activity. The target victim user has a linked AppleID account. We use the `FSEvents` API [3] which requires directory paths to watch as an input when creating an `FSEvents` stream.

Temporal Resolution. Using the same methodology as Linux and Windows, we measure `FSEvents` temporal resolution across 1 000 attempts using two programs, yielding an average of 11.48 ms (n=1000, σ=0.57 ms), the slowest of the three operating systems. The increase in CPU usage is 0.002 6 % according to `htop`.

6.2 User, Application, and System Behavior

Applying our templating to cross-user monitoring, we find user-specific activity confined to home directories, as on Linux and Windows. Still, system-level operations raise file notifications via shared system files leaking user, application, and system behavior.

Settings and Connectivity Interaction. When an AppleID-user opens the System Settings (application), we observe modifications to the system-wide `exists.plist` file. Cycling through audio output and input options generates events to the `SystemSettings-.plist` file. Modifying power settings (such as "Preventing automatic sleep when the display is off") update the system-wide power management configuration file in `PowerManagement.plist`.

Bluetooth activation triggers temporary keychain file changes in `/Library/Keychains/`, though this is not exclusive to Bluetooth. Adding or removing Bluetooth devices updates `MobileBluetooth-.devices.plist`. Plugging or unplugging a network cable instantly updates DNS configuration in `resolv.conf`. Printing operations, e.g., opening dialogs, adding printers, and print jobs, update world-readable files in the `cups` directory. External storage insertion or removal creates or deletes mount points in `/Volumes/`.

Application Behavior. Application installs, updates, and removals are detectable via changes in `/Applications/`, with App Store installations additionally producing temporary sandboxes and logs. Media apps (Podcasts, Voice Memos, Books) update `SystemSettings.plist` on launch and exit. Chrome triggers attribute modifications in `Google Chrome.app/`. Apple iWork applications (Pages, Numbers, Keynote) produce patterns when saving files. Steam generates temporary files in `/private/tmp/` during game launches and exits. VMware Fusion reveals virtualization activity in detail, due to the per-VM unique hash-based identifier while running, removed upon pause or shutdown, enabling per-VM state tracking.

7 Discussion

In this paper, we demonstrated the first generic file-notification template attacks on Linux, Windows, and macOS, allowing an unprivileged attacker to infer user, system, and application behavior.

On **Linux**, the unreadable file bypass (Section 4.2.1) enables access notifications to device files, allowing for local and remote inter-keystroke timing attacks. Linux is also the only system with reliable read-only file access notifications, making it the leak the most of the three systems. As **macOS** is unaffected by these two factors, we see far less information leakage. On **Windows**, the full file path leakage is undocumented but by-design.[1] Although Microsoft responded to our responsible disclosure that this behavior "only reveals file names and paths within another user's profile directory [and] does not expose file contents or sensitive data", we demonstrate that knowing *only* the file names and paths within another user's directory can have serious security and privacy implications. Firefox's storage design leaks visited websites with a 97.8 % F_1 score (no false positives), and even Chromium's more conservative design

yields a 48.5 % F_1 score with no false positives. Consequently, we believe Microsoft should reconsider this concerning design choice.

All systems leak some global information. However, macOS leaks no additional private information as it only exposes globally-readable file info; there are no additional unreadable-file bypasses like Windows, Linux, or Android, *i.e.*, private directories cannot be monitored. While the user, application, and system behavior leakage in each OS may seem less substantial than the unreadable-file bypasses, it still yields meaningful insight into user activity.

Limitations. A general limitation of side-channel attacks, file-notification attacks suffer from incorrect interpretations, such as multiple actions producing identical notifications, e.g., reading versus executing a binary. However, error rates can be reduced by watching more linked files associated with an action or with repeated observations, thus deriving more precise conclusions. Watching large file sets may impact performance like `kqueue` on macOS [14], and Linux caps watched files from 65 536 to 524 288 on our systems. However, these limits only affects templating. Otherwise, file-notification attacks are stealthy and memory efficient.

Countermeasures. In early 2026, Linux deployed a mitigation against the unreadable-file bypass on *device* files only. All other files on the system are still susceptible to the bypass, and as such, we propose two capability checks: monitoring one's own files, and any readable file. On Windows, we propose disallowing the monitoring of entire drives. On Windows and macOS, the kernel could introduce a permission system (for context, access control, owned files and directories, minifilters). Sandboxing techniques may also prevent file-operation events *within* the sandbox, e.g., Docker [13, 54].

Related Work. Closest to our work are two studies on Android's `FileObserver` class that looked specifically at UI updates [1] and app startups [67]. However, they restricted themselves to these scenarios and the Android platform, and did not explore additional event leakage for other attacks or platforms. In contrast, we investigate multiple operating systems, showing the much larger impact of file notification leakage while uncovering several questionable design choices in Linux and Windows' file notification APIs. Furthermore, we extend prior work [67] by showing the private folder monitoring applies to *internal* storage, that concrete files (and not just folders) are exposed, and, most importantly, that `FileObserver` bypasses per-app restrictions on shared external storage entirely.

File accesses can also be monitored via page cache attacks [6, 18, 57, 70]. Such attacks have a spatial resolution of a single memory page. Despite having file-grained resolution, our attack requires no page cache eviction and, more importantly, carries detailed information, e.g., open or close, compared to just a page *access*.

8 Conclusion

We presented the first generic file-notification template attacks on Linux, Windows, and macOS. We demonstrated that the attack can achieve a high resolution in the range of 0.5 ms to 11.5 ms enabling numerous attacks, e.g., an inter-keystroke timing attack with 100 % F_1 scores, an authentication UI redress attack on KDE Plasma 6, and a website fingerprinting attack on the top-100 open-world websites with an F_1 score of 87.9 %. In particular on Windows, we uncovered a serious design decision where unprivileged users can leak the paths of *all* files even without read permissions on the containing directory, leading to a direct leak of user and application file usage, allowing to monitor websites with an F_1 score of 97.8 % on the top-1000 websites. We conclude that file-notification attacks are a generic security problem, cross-cutting across all modern operating systems despite their independent development. Future operating systems should account these insights and deploy low cost mitigations.

Acknowledgments

We thank our anonymous reviewers across conferences for their feedback and guidance. We additionally would like to thank Fabian Rauscher, Li-Chung Chiang, Lukas Giner, Simone Franza, and Theresa Dachauer for their (literal and metaphorical) inputs at various stages of this work. We thank Amir Goldstein, Jan Kara, and Greg Kroah-Hartman for handling our Linux kernel disclosure and for their work on patches that partially mitigate our findings on Linux. Long after our submission and despite our report to Microsoft, we only became aware of CVE-2025-27738 and CVE-2025-21197 much later, which are similar to our Windows findings in Section 5. These were reported to Microsoft by Sébastien Huneault in April 2025. Microsoft's "Access check enhancements to prevent unauthorized disclosure of file paths" update (KB ID: 5058189) introduces a registry policy, `EnforceDirectoryChangeNotificationPermissionCheck`, that mitigates the behavior we report. This policy is disabled by default, *i.e.*, all the attacks we report in this paper work out-of-the-box on Windows systems.

LLMs were used for minor editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality; in particular, Copilot, ChatGPT, Claude, and Grammarly. No passages were copied without full review. Additionally, the listed tools were utilized for code development. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Woo Hyun Ahn, Sanghyeon Park, Jaewon Oh, and Seung-Ho Lim. 2016. Inishing: A UI Phishing Attack to Exploit the Vulnerability of Inotify in Android Smartphones. *IEICE Transactions on Information and Systems* 99-D, 9 (2016), 2404–2409.

[2] Apple. 2000. KQUEUE(2). https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/kqueue.2.html

[3] Apple. 2025. File System Events. https://developer.apple.com/documentation/coreservices/file_system_events

[4] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2022. DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels. In *FAST*.

[5] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2017. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. *Cryptology ePrint Archive, Report 2017/968* (2017).

[6] Novak Boskov, Naor Radami, Trishita Tiwari, and Ari Trachtenberg. 2022. Union Buster: A Cross-Container Covert-Channel Exploiting Union Mounting. In *International Symposium on Cyber Security, Cryptology, and Machine Learning*.

[7] Billy Brumley and Risto Hakala. 2009. Cache-Timing Template Attacks. In *AsiaCrypt*.

[8] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. 2002. Template attacks. In *CHES*.

[9] Congcong Chen, Jinhua Cui, Gang Qu, and Jiliang Zhang. 2024. Write+Sync: Software Cache Write Covert Channels Exploiting Memory-disk Synchronization. *TIFS* (2024).

[10] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks.. In *USENIX Security*.

[11] ClamAV Documentation. 2026. On-Access Scanning. https://docs.clamav.net/manual/OnAccess.html

[12] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. 2018. Observations on typing from 136 million keystrokes. In *CHI Conference on Human Factors in Computing Systems*.

[13] Docker. 2025. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/

[14] Enrico Maria Crisostomo. 2025. fswatch. https://github.com/emcrisostomo/fswatch/tree/1.18.3

[15] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In *S&P*.

[16] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. 2022. Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* (2022).

[17] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS*.

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

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[20] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.

[21] Cheng Gu, Yicheng Zhang, and Nael Abu-Ghazaleh. 2025. I know What You Sync: Covert and Side Channel Attacks on File Systems via syncfs. In *S&P*.

[22] Andrew Hintz. 2003. Fingerprinting Websites Using Traffic Analysis. In *PETS*.

[23] Michael Augustus Hogye, Christopher Taddeus Hughes, Joshua Michael Sarfaty, and Joseph David Wolf. 2001. *Analysis of the Feasibility of Keystroke Timing Attacks over SSH Connections.* Technical Report. School of Engineering and Applied Science University of Virginia.

[24] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *S&P*.

[25] Qisheng Jiang and Chundong Wang. 2024. Sync+Sync: A Covert Channel Built on fsync with Storage. In *USENIX Security*.

[26] Jsparber. 2024. Understanding GNOME Shell's focus stealing prevention. https://blogs.gnome.org/shell-dev/2024/09/20/understanding-gnome-shells-focus-stealing-prevention/

[27] Jonas Juffinger, Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss. 2025. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In *NDSS*.

[28] Jonas Juffinger, Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss. 2025. The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer. In *DIMVA*.

[29] Michael Kerrisk. 2014. Filesystem notification, part 1: An overview of dnotify and inotify. https://lwn.net/Articles/604686/

[30] Paul Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*.

[31] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology (CRYPTO)*.

[32] Jiri Kosina. 2019. make mincore() more conservative. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=134fca9063ad4851de767d1768180e5dede9a881

[33] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. 2023. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In *USENIX Security*.

[34] Linux. 2024. fsnotify.c. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/notify/fsnotify.c?h=v6.12

[35] Linux. 2025. *fanotify(7) — Linux manual page.* https://www.man7.org/linux/man-pages/man7/fanotify.7.html.

[36] Linux. 2025. *fanotify_init(2) — Linux manual page.* https://man7.org/linux/man-pages/man2/fanotify_init.2.html.

[37] Linux. 2025. *F_NOTIFY(2const) — Linux manual page.* https://man7.org/linux/man-pages/man2/F_NOTIFY.2const.html.

[38] Linux. 2025. *inotify(7) — Linux manual page.* https://www.man7.org/linux/man-pages/man7/inotify.7.html.

[39] Linux. 2025. *pts(4) — Linux manual page.* https://www.man7.org/linux/man-pages/man4/pts.4.html.

[40] Linux. 2025. *tty(4) — Linux manual page.* https://www.man7.org/linux/man-pages/man4/tty.4.html.

[41] Linux Kernel. 2005. Changelog 2.6.13. https://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.13

[42] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[44] LSB Workgroup, The Linux Foundation. 2015. Filesystem Hierarchy Standard Version 3.0. https://refspecs.linuxfoundation.org/fhs.shtml

[45] lsyncd/lsyncd. 2024. Lsyncd – Live Syncing (Mirror) Daemon. https://github.com/lsyncd/lsyncd

[46] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2024. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security*.

[47] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. 2025. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In *NDSS*.

[48] MaK Ulac. 2025. GNOME 50: Wayland-Only Brings Enhanced Security and Isolation. https://linuxsecurity.com/news/desktop-security/gnome-50-wayland-linux-security

[49] Microsoft. 2022. ReadDirectoryChangesW function (winbase.h). https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-readdirectorychangesw

[50] Microsoft. 2024. FindFirstChangeNotificationA function (fileapi.h). https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-findfirstchangenotificationa

[51] Microsoft. 2024. SECURITY_DESCRIPTOR structure (winnt.h). https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-security_descriptor

[52] Microsoft. 2025. FileSystemWatcher Class. https://learn.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher

[53] microsoft/vscode. 2024. File Watcher Internals. https://github.com/microsoft/vscode/wiki/File-Watcher-Internals

[54] moby/profiles. 2025. seccomp/default.json: Default Seccomp Policy. https://github.com/moby/profiles/blob/main/seccomp/default.json

[55] Patrick Mochel and Mike Murphy. 2011. sysfs - The filesystem for exporting kernel objects. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/sysfs.txt?h=v3.2

[56] John Monaco. 2018. SoK: Keylogging Side Channels. In *S&P*.

[57] Sudheendra Raghav Neela, Jonas Juffinger, Lukas Maar, and Daniel Gruss. 2026. Eviction Notice: Reviving and Advancing Page Cache Attacks. In *NDSS*.

[58] Binh Nguyen. 2004. Linux Filesystem Hierarchy: /dev. https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/dev.html

[59] NIST National Vulnerability Database. 2007. CVE-2007-0843. https://nvd.nist.gov/vuln/detail/CVE-2007-0843 ReadDirectoryChangesW permission check.

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

[61] Eric Paris. 2009. fanotify: the fscking all notification system. https://lwn.net/Articles/339253/

[62] Vojtech Pavlik and Stephen Kitt. 2022. *EVTEST(1) — Debian manual page.* https://manpages.debian.org/testing/evtest/evtest.1.en.html.

[63] polkit. 2025. *polkit.* https://polkit.pages.freedesktop.org/polkit/.

[64] Mufan Qiu, Lihsuan Chuang, Dohhyun Kim, Huaizhi Qu, Tianlong Chen, and Andrew Kwong. 2026. KeyTAR: Practical Keystroke Timing Attacks and Input Reconstruction. In *S&P*.

[65] Christian Rechberger and Elisabeth Oswald. 2004. Practical template attacks. In *WISA*.

[66] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. 2017. Automated Website Fingerprinting through Deep Learning. In *NDSS*.

[67] Antonio Ruggia, Andrea Possemato, Alessio Merlo, Dario Nisi, and Simone Aonzo. 2023. Android, Notify Me When It Is Time To Go Phishing. In *EuroS&P*.

[68] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. 2010. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *4th USENIX Conference on Offensive Technologies*.

[69] Michael Schwarz, Florian Lackner, and Daniel Gruss. 2019. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *NDSS*.

[70] Martin Schwarzl, Erik Kraft, and Daniel Gruss. 2023. Layered Binary Templating. In *ACNS*.

[71] Narasimha Shashidhar and Dylan Novak. 2015. Digital Forensic Analysis on Prefetch Files. *International Journal of Information Security Science* 4, 2 (2015), 39–49.

[72] Chaoqun Shen, Jiliang Zhang, and Gang Qu. 2023. MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization. In *DAC*.

[73] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security*.

[74] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory Deduplication as a Threat to the Guest OS. In *EuroSec*.

[75] FileCAB Team. 2019. Disabling Last Access Time in Windows Vista to improve NTFS performance. https://techcommunity.microsoft.com/blog/filecab/disabling-last-access-time-in-windows-vista-to-improve-ntfs-performance/423328

[76] The Android Open Source Project. 2020. FileObserver.java. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/java/androi

d/os/FileObserver.java
[77] The Android Open Source Project. 2026. Scoped storage. https://source.android.com/docs/core/storage/scoped
[78] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. 2023. Covert-channels in FPGA-enabled SmartSSDs. *ACM Transactions on Reconfigurable Technology and Systems* (2023).
[79] Pepe Vila and Boris Köpf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security*.
[80] Han Wang, Hossein Sayadi, Avesta Sasan, P D Sai Manoj, Setareh Rafatirad, and Houman Homayoun. 2021. Machine Learning-Assisted Website Fingerprinting Attacks with Side-Channel Information: A Comprehensive Analysis and Characterization. In *International Symposium on Quality Electronic Design (ISQED)*.
[81] Hannes Weissteiner, Tobias Weiser, Roland Czerny, Sudheendra Raghav Neela, Fabian Rauscher, Jonas Juffinger, and Daniel Gruss. 2026. FROST: Fingerprinting Remotely using OPFS-based SSD Timing. In *DIMVA*.
[82] Boyuan Yang, Ruirong Chen, Kai Huang, Jun Yang, and Wei Gao. 2022. Eavesdropping User Credentials via GPU Side Channels on Smartphones. In *ASPLOS*.
[83] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[84] Zed. 2026. Zed on Linux. https://zed.dev/docs/linux
[85] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security*.

A Open Science

We make our code on Linux, Android, Windows, and macOS conveniently available at: https://github.com/isec-tugraz/file-notification-attacks.

B Ethical Consideration

We responsibly disclosed our findings to the security teams of Linux, Android, Windows, and macOS between August to October 2025. All four parties acknowledged our findings and we worked towards mitigation strategies with the Linux security team in December 2025; subsequently, partially-mitigated patches have been deployed in early 2026 and our findings were assigned CVE-2025-68788. We disclosed the focus-stealing attack on authentication dialogs (see Section 4.4.3) to KDE's security team in November 2025, who have also acknowledged our findings. The KDE security team replied that focus-stealing prevention is not meant as a security measure, but rather to avoid race conditions with annoying popups. In our emails with KDE, we did not mention the Linux `inotify` side channel and instead assumed that the authentication dialog is already active.

All our experiments were performed on local systems and did not cause interference to others. Our website fingerprinting experiments on Linux with the top-100 websites required us to visit each website 30 times in total. The 750 open-world websites were visited once. There was no interaction on these websites apart from only visiting it (like normal users), and we did not cause any unusual traffic. For monitoring website visits on Windows, our script visited the top-1000 websites two times in total.

C Cross-Docker-Container File Notifications

In docker, we find that reading a small set of files in a container trigger modify notifications in another file. Interestingly, we find that this notification is sent *between* containers, and surprisingly even of different images. We tested our findings on two host systems Ubuntu 22.04 (AMD Ryzen 7 Pro 5850U, kernel 6.8.0) and 24.04 (Intel Core i7-1260P, kernel 6.8.0) and found leakage between two images based on different Linux distributions. In particular, we tested combinations of Ubuntu 22.04.5, Debian 12, and Fedora 43. These files are located in tree of `/sys/devices/pci0000:00/`, but the specific numbers differed between the two host systems. We consider the specifics on our Ubuntu 22.04 system with the files located in the tree of `/sys/devices/pci0000:00/0000:00:08.1/0000:05:00.4/usb6`.

We find that reading any of the four files at `6-1/`**`6-1:1.0`**`/6-1--port[1-4]/disable` (file Ⓐ) sends a modify event to **`6-0:1.0`**`/-usb6-port1/state` (file $Ⓐ_{\mathcal{N}}$). Similarly, reading any of the four files at `6-1/`**`6-1.2`**`/6-1.2:1.0/6-1.2-port[1-4]/disable` (file Ⓑ) sends two modify events, one to the previous state file $Ⓐ_{\mathcal{N}}$ and another one to **`6-1`**`/6-1:1.0/6-1-port2/state` (file $Ⓑ_{\mathcal{N}}$). Although the temporal resolution of `inotify` is less than 10 ms on our system (see Section 4.2.6), we find that reading the `disable` files requires discernible time (up to 0.25 s).