Leaky Apps: Targeted Deanonymization on Mobile Phones

Robert Blacha
New Jersey Institute of Technology
Newark, USA

Yossi Oren
Ben-Gurion University of the Negev
Be'er Sheva, Israel

Reza Curtmola
New Jersey Institute of Technology
Newark, USA

Abstract

Targeted Deanonymization attacks allow an attacker who controls a website to infer the identity of specific target users browsing that website. They are a severe privacy risk, as they can then be used to carry out highly-targeted attacks against the inferred identities.

These attacks were previously shown to be practical in the desktop environment. In this work, we set to investigate the feasibility of targeted deanonymization in a mobile setting, which presents new opportunities but also new challenges for the attacker. We discover a surprising reality: The attack surface for targeted deanonymization on mobiles is larger than in the desktop setting. We replicate successfully on the Android system all the scenarios that were possible in the desktop setting, and also introduce new attack variants specific to the mobile setting. Notably, the *app pop-up* variant leverages Android intents to bypass the need for web cookies altogether.

We present a decision tree that the attacker can navigate to select the appropriate attack variant, depending on the specific configuration on target user's mobile device. We show that the attacker can target an overwhelming majority of mobile users, including multiple mobile browsers, in-app browsers embedded into common apps, and many resource-sharing services, with attack accuracies and times comparable to those in the desktop setting. We also discuss defenses specific to the mobile setting, ranging from those that can be enabled by users to those that can be deployed by app developers, resource-sharing services, and mobile OS and browser vendors.

CCS Concepts

• **Security and privacy** → *Browser security*; *Web application security*.

Keywords

Web privacy; side-channel attacks; mobile phones

ACM Reference Format:
Robert Blacha, Yossi Oren, and Reza Curtmola. 2026. Leaky Apps: Targeted Deanonymization on Mobile Phones. In *Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy (CODASPY '26), June 23–25, 2026, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3800506.3803502

1 Introduction

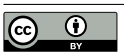
Targeted deanonymization (TD) attacks [27, 28, 35, 36] allow an attacker who controls a website to learn whether a specific target user is browsing the website, by correlating the web session with a well-known public identifier of the target, such as their email address or social media handle. These attacks pose a serious challenge to the assumption of privacy on the web, which is critical for vulnerable populations such as activists, journalists, and whistleblowers. Furthermore, such attacks can also be used as a stepping stone towards executing more sophisticated attacks, such as deploying a zero-day exploit against the identified target [36].

The workflow of a TD attack is presented in Fig. 1. To mount the attack, the attacker only needs to know the target through a public identifier, such as an email address, a LinkedIn user identifier, or an Instagram handle. The attack then leverages *state-dependent URLs (SD-URLs)* — URLs which return different responses depending on a user's identity. An example of such SD-URLs are *leaky resources* [27, 35] such as images or videos, which are stored at resource-sharing services such as YouTube, Google Drive or Dropbox. The attacker creates an SD-URL correlated with the target's public identifier, for example by sharing a YouTube video with the target's Gmail address, and places it on an attack website together with code that collects a side-channel trace. Users are then induced to visit the attack webpage, for example through a phishing link or an embedded advertisement. If a user currently visiting the attack website can access the embedded resource, then this visitor is the intended target.

The same origin policy should ostensibly prevent the attack website from learning whether the resource was loaded successfully or not. Several works, however, have shown how this protection can be circumvented. Zaheri et al., in particular, showed that attackers can break cross-site isolation and perform TD by using JavaScript-based side-channel attacks[1] [36]. Zaheri et al. showed that the combination of cache attacks and a machine learning pipeline enables the attacker to execute the attack efficiently and scalably. Since side-channel attacks exploit fundamental hardware characteristics, they are much harder to defend against [5, 6]. In particular, side-channel attacks work even if the resource-sharing service does not allow the embedding of its resources, or when browsers disable third-party cookies for embedded resources.

A common aspect in all prior work on targeted deanonymization attacks is the focus on the *desktop environment*. In this setting, we assume web content is typically consumed through the system default web browser. This implies that the system default browser's "cookie jar" contains authentication cookies for the resource-sharing service. Thus, when the attack page loads an SD-URL from the resource-sharing service, it receives different content, depending on the existence of these cookies, enabling targeted deanonymization.

This assumption, however, does not necessarily hold in the *mobile setting*, which is the focus of this work. In contrast to desktop computers, mobile users do not typically use the default web



CODASPY '26, Frankfurt am Main, Germany

ACM ISBN 979-8-4007-2562-3/2026/06
https://doi.org/10.1145/3800506.3803502

[1] The attack can also be executed via cross-site leaks (XS-leaks) [28], which are mechanisms that exploit behaviors that bypass the same origin policy, such as status code leaks, page content leaks, and header leaks [15]. XS-leaks are being gradually patched by browser vendors and/or resource-sharing providers as they are being discovered.

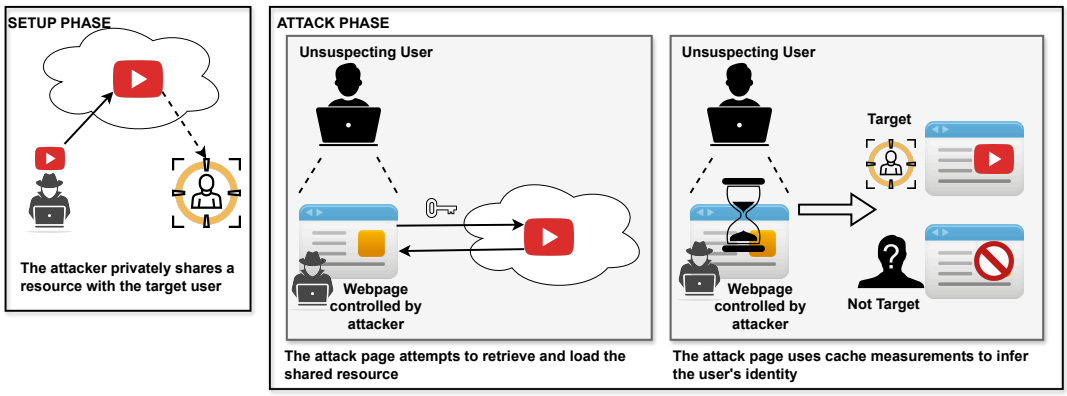


Figure 1: Workflow of a targeted deanonymization attack. In the Setup phase, the attacker privately shares a resource with the target user. In the Attack phase, once a user gets lured to visit the attack page, the page attempts to retrieve and play the shared resource. At the same time, the page takes cache measurements and uses them to decide if the current user is the target.

browser for all their Internet activities; instead, they often use dedicated mobile apps, such as Facebook or Instagram, to access social media and resource-sharing services. Social media apps tend to "keep" their users in the app, even as they click on links, by rendering web content using an *in-app browser (IAB)*, which is a full browser environment inside the app. When users exit this captive web session, they return to the app. Clearly, users are not expected to log into resource-sharing services through these in-app browsers, which are typically used for short, ephemeral browsing sessions. Furthermore, some users do not log into their resource-sharing services through the system default browser app at all; instead, they only use dedicated apps, such as the Facebook app, to access these services. This reality challenges the attack workflow in two main ways: first, if a URL is shared with the target through a mobile app, for example through a private message on Instagram, it is not clear that the system default browser will be used to open the URL; instead, it may be opened in the IAB. Second, even if the URL is opened in the system default browser, it is not clear that the browser's cookie jar will contain authentication cookies for the resource-sharing service, since the user may not have logged into the service through the browser. As a result, it is tempting to think that users are less vulnerable to TD attacks in a mobile setting.

In this work, we set to investigate the feasibility of targeted deanonymization attacks in a mobile setting, focusing on the Android operating system. We discover a surprising reality: the mobile setting is, in fact, even more vulnerable to such attacks than the desktop setting. More specifically, we discover that the in-app browser of many popular mobile apps, including social media apps such as LinkedIn, Teams, Discord, Snapchat and Slack, shares its cookies with the system default browser. Thus, the "leaky resource" attacks described in the desktop setting can also be executed in this IAB with minimal modification. An exacerbating factor is that Android users are all but forced to log into their Google account in the system default browser, implicitly logging them into YouTube, Google Drive, and other Google services as well. We note that users cannot uninstall the Chrome and YouTube apps on an Android phone, which puts all Android users at risk.

In addition, we discover than in many settings the attacker can completely bypass the need for cookies: We describe a special mobile-specific variant of the TD attack called the *app pop-up* variant, in which the attacker page can abuse the Android *intent* system to cause a specific app to open an SD-URL, instead of the system default browser. Through this variant, mobile TD attacks can effectively be run against many types of public identifiers, including Facebook or Instagram handles, even if the user never logged into the resource-sharing service through any browser.

A summary of our results is presented in Table 1, assuming the default Chrome browser on a stock Android phone. As the Table shows, all of the shared resources we tested can be exploited through multiple delivery vectors.

We introduce three variants of the targeted deanonymization attack on mobile devices. Each variant covers a specific mobile environment that may be present on the target user's mobile device. Based on information about the web environment in which the attack page is loaded, the attacker navigates a decision tree and executes one of these variants.

The first attack variant, referred to as the *embedding variant*, embeds the leaky resource into the attack page using an `<iframe>`. This variant, which assumes the user has logged in to the resource-sharing service using the system browser, is suitable for mobile browsers that allow third-party cookies by default such as Chrome and requires no additional interaction from the user beyond navigating to the attack webpage.

The second attack variant, referred to as the *app pop-up variant*, uses *Android intents* to open the leaky resource directly in a social media app, while measuring the side-channel trace using an attack webpage. Uniquely, this attack variant does not require the user to be logged into the resource-sharing service through any browser, as it leverages the app's native authentication.

The third attack variant, known as the *downgrade variant*, presents the user with the option to open the attack page in a different browser, preferably one that allows third-party cookies. This essentially downgrades the browsing environment to enable the embedding attack variant; it is suitable for the Tor browser, which blocks third-party cookies and does not allow opening new windows.

In summary, this work makes the following contributions:

	SMS/MMS	Gmail (Emails, Google Chat)	LinkedIn (Posts, Comments, Bios, Ads)	Microsoft Teams (Mesages)	Discord (Messages, Server Posts)	SnapChat (Messages, Ads)	Slack	Yahoo Finance	Google Discover	Reddit (Posts, Comments, Ads)	Instagram (Story, Bio, Ads, Messages)	Facebook/ Facebook Messenger	TikTok (Business Accounts)
YouTube/Google Drive video	●	●	●	●	●	●	●	●	●	●	◐	◐	◐
DropBox video	●	●	●	●	●	●	●	●	●	●	◐	◐	◐
LinkedIn video post	●	●	○	●	●	●	●	●	●	●	◐	◐	◐
OneDrive video	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
Instagram profile	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	◐	◐
Facebook video post	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	◐
Reddit video post	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	◐	◐	◐

Table 1: Exploitability of targeted deanonymization attacks for different shared resources (rows) and delivery vectors (columns). ●: exploitable with no interaction; ◐: exploitable but requires further interaction; ○: not exploitable. We assume a stock Android phone is used, with Chrome as the default browser app. We also assume that the target is logged in their default browser app (Chrome) into the resource-sharing service leveraged for the attack.

– We study the feasibility of cache side channel-based targeted deanonymization attacks [36] in the mobile environment. This environment is much more fragmented than the desktop environment, containing a complex ecosystem of apps, mobile browsers, and in-app browsers. While this increases the mobile attack surface, it also makes it more challenging to trigger the loading of the shared resource, since mobile users have a variety of ways to access shared resources. Based on an extensive exploration of the attack surface on mobile devices, we identify and systematize relevant features available on popular mobile browsers (including in-app browsers) and on popular social media apps that can be leveraged to execute the TD attack. We also identify popular resource-sharing services that expose leaky resources which can be leveraged for the attack. Putting all these together, we introduce three attack variants, each of which can be effective depending on the target's specific configuration (browser/in-app browser and leaky resource): *embedding* variant, *app pop-up* variant, and *downgrade* variant.

– We show how the attacker can use a decision tree algorithm to dynamically generate the most effective attack page based on information that can be practically extracted from the victim. This allows the attacker to choose the most effective of the three attack variants, depending on the victim's choice of mobile browser (Chrome, Firefox, DuckDuckGo, Edge, Brave, Opera, Tor), in-app browsers of popular social media apps (Gmail, LinkedIn, Teams, Discord, SnapChat, Slack, Yahoo Finance, Google Discover, Facebook, Facebook Messenger, Instagram, TikTok, Reddit), and resource-sharing services (YouTube, Google Drive, DropBox, OneDrive, LinkedIn, Instagram, Facebook, Reddit). Overall, the attacker can target an overwhelming majority of mobile users.

– We experimentally demonstrate practical end-to-end attacks for the three attack variants, all of which run in under 3 seconds. We also compare the attack's stealthiness and feasibility among the three attack variants.

– We discuss several defenses that are specific to the mobile setting, ranging from those that can be enabled by users to those that can be deployed by app developers, resource-sharing services and mobile OS and browser vendors.

2 Background

2.1 CPU Cache-based Side-Channel Attacks

Cache attacks leverage the fact that all processes compete for the limited space available in the CPU cache. A malicious process can exploit this contention to bypass software-based isolation mechanisms and infer information about the internal state of other processes. A popular approach to execute such attacks is based on the **Prime+Probe technique** [21, 23]. In this method, the attacker first "primes" the cache by loading their own data into specific cache sets, which are regions of the cache that are shared between the attacker and the victim processes. After the victim process executes, the attacker "probes" these sets to detect whether their data has been evicted. A low access time indicates the attacker's data is still in the cache, whereas a high access time means that it was evicted and replaced by the victim's data. By repeating this process, the attacker can infer the victim's memory access patterns, potentially learning sensitive information such as encryption keys.

Prime+Probe attacks require nanosecond-level measurements to distinguish between cache hits and misses. Such timer resolution is not typically available through JavaScript. Instead, the **Cache Occupancy** variation [26] was introduced to enable the attack in web browsers. The attacker allocates a large buffer that covers the entire last-level cache (LLC) and accesses this buffer in the prime step, bringing the entire LLC into a known state. In the probe step, the attacker measures the time required to access the entire buffer (i.e., the *buffer access time*). Cache contention caused by victim process evicting the attacker's buffer introduces measurable delays when reading this buffer, allowing the attacker to detect victim activity. By reading a large buffer, the cache occupancy attack can

be executed based on coarse-grained timers which are available in web browsers (i.e., millisecond accuracy).

Some browsers, such as the highly-secure Tor browser, only provide an even coarser-grained timer of 100ms. **Sweep Counting** [25] is a modification of the cache occupancy attack, designed to work in such scenarios. The attacker counts the number of times it can access the entire buffer in a fixed interval of time, rather than measuring the time needed to read the buffer once.

2.2 The Targeted Deanonymization Attack

A targeted deanonymization attack [15, 27, 28, 31, 35] allows an attacker who controls a webpage to infer if a specific individual (*i.e.*, the target) is browsing that webpage. The attacker only needs to know a public identifier of the target, such as an email address or a social media handle. The attacker leverages the fact that the victim has an account with a resource-sharing service that exposes *state-dependent URLs (SD-URLs)* to reference its shared resources. When a user makes a request to a state-dependent URL, the response depends on the user's identity, allowing the attacker-controlled page to infer the user's unique identity.

Many services were shown to enable target deanonymization attacks, including generic storage sites, media sharing sites, code-hosting repositories and social media sites. It is quite common for users to remain logged into such services, and in particular to the Google/YouTube ecosystem, for extended periods of time.

As illustrated in Fig. 1, the attack proceeds in two phases. In the setup phase, the attacker uploads a resource to the resource-sharing service, and then binds it to the victim's identity. This binding can be performed using one of two approaches. Notably, for both of these approaches, the binding occurs stealthily, as the target does not typically get a notification. In the *sharing-based approach* [27], the attacker privately shares the resource with the target. This works for services which allow to privately share their resources, such as YouTube, Google Drive and DropBox. In the *blocking-based approach* [31], the attacker makes the resource public and then blocks the target from viewing any resources owned by the attacker. This works for services which do not allow to privately share their resources or ignore cookies from cross-site requests, such as LinkedIn, Instagram and Facebook. The attacker also creates an *attack webpage* that attempts to load the resource.

In the attack phase, the attacker first lures a user to visit the attack page, and then determines if the resource loads or not. The user is deemed to be the target depending on whether the resource was successfully loaded (in the sharing-based approach) or was not successfully loaded (in the blocking-based approach).

The attacker page cannot trivially determine the outcome of loading the resource, as this constitutes a cross-origin information leak, which should be normally prevented by browser security policies [11, 12]. Such policies, however, have been shown to be difficult to implement in practice, and can be bypassed by exploiting software-based cross-site leak bugs (XS-leaks) [15, 27, 28, 35].

Simply fixing software-based XS-leaks is not enough to prevent TD attacks: The attack can still be executed by using cache-based side channels. Because these attacks operate on hardware-level properties of the victim's computer, they disregard any software-imposed boundaries [36]. Specifically, while the attack page is rendered by a user's browser, the page can use the cache occupancy method to determine if the resource was loaded. Essentially, longer buffer access times indicate the shared resource was loaded.

3 Attack Techniques

3.1 Threat Model

Following the established threat model for targeted deanonymization, we assume that there is a *target user*, known to the attacker by a public identifier such as an email address, a LinkedIn user identifier, or an Instagram handle. The attacker is capable of luring the users to a website under its full or partial control, and furthermore has the ability to inject JavaScript code into this website. The attacker seeks to discover whether the user currently browsing this website is the target [27, 35, 36].

In this specific work, we assume the target has an Android mobile device. We further assume that the target has an account with a resource-sharing service that exposes leaky resources, has installed on their mobile device the dedicated app for this service, and has logged in to the service using this app. Such services include, but are not limited to, Gmail (which implies accounts with services such as YouTube and Google Drive), Dropbox, OneDrive, LinkedIn, Instagram, Facebook, or Reddit. We note that the attacker does not need to control the resource-sharing service that is leveraged to execute the attack, but only to be registered as a user of the service.

3.2 How do Users Access Resource-Sharing Services on Android Devices?

In the desktop setting, all online content is typically consumed through a single web browser application. The "cookie jar" of this desktop browser naturally contains authentication tokens that authenticate the user to all online services, including resource-sharing services. On mobile devices, in contrast, users access online content in a variety of ways.

The System Default Browser. On every Android phone, a single app is designated as the default browser app, and is used to open web pages by default. On the vast majority of Android devices, this is the default Google Chrome browser app. Users can, however, install alternative browsers and configure them as the default browsers. Examples of these alternative browsers include Firefox, DuckDuckGo, Brave, Opera, Edge, and Tor. An important difference among these browsers is their support for *third-party cookies*, a factor which affects the feasibility of the targeted deanonymization attack, as we will see later.

In-App Browsers. Many mobile apps, especially social media apps, prefer that the user does not leave the app when accessing web content. Android supports this requirement by allowing apps to implement what is called an *in-app browser (IAB)*, which is essentially a web browser embedded inside the app and sharing its user interface. In these apps, whenever a user clicks on a link, the app opens the link in the IAB, and when the user closes the IAB or hits the "Back" navigation button, they are returned to the parent app.

Android offers two general approaches to implement an IAB, each with its own characteristics that affect the feasibility of the targeted deanonymization attack. The first approach is to use the *Custom Tabs* feature, which creates an instance of the system's

App Name	IAB Type	Cookie Jar
Gmail, LinkedIn, Teams, Discord, SnapChat, Slack, Yahoo Finance, Google Discover	Custom Tabs	Shared
Reddit	Web Embed	Shared
Instagram, Facebook, Facebook Messenger, TikTok	Specialized	Private

Table 2: IABs for popular social media apps.

default browser app and embeds it inside the parent app. Significantly for our attack, a Custom Tabs IAB shares cookies and other authentication tokens with the default browser app[2]. Some apps, such as Reddit and Groupon, do not directly use a Custom Tabs implementation for their IAB. Instead, the app creates a wrapper for a browser to run in, from which it can then implement its own browser or instantiate the default browser. This approach allows an app to have more control and to limit traffic from leaving the app. The Reddit embedded browser, for example, treats opening new windows as redirection and ignores sending Android intents. We refer to these as "Web Embed" IABs, and note that they also share their cookies with the default browser app. The second approach is the *specialized* approach. Here, the app implements its own IAB, typically based on the Android WebView component, allowing it finer-grained control over the content being displayed. In the specialized approach, the IAB maintains its own cookie jar. Table 2 describes the types of IABs used by popular social media apps.

Dedicated Apps. Many resource-sharing services provide dedicated mobile apps that users can install on their devices. This is the most common and natural way for users to access social media services such as Instagram, Facebook, TikTok, and Reddit. These apps typically authenticate themselves against the online service, independently of the installed browser app and its cookie jar. Thus, even if a user has not authenticated against an online service using the default browser app, the user may still be exposed to targeted deanonymization, a fact that we will exploit later.

Intents and Intent Filters. An important component in our attack is the *intent filters* feature in Android. This feature allows an app to register itself at installation as the system's default way of opening resources following a certain URI pattern. The most common filters are for `https` *intents*: Apps that handle a specific type of resource define an `https` intent for that type of resource. The YouTube app, for example, registers itself to handle all URLs starting with `youtube.com`, `youtu.be`, and `m.youtube.com`. When a user attempts to open a URL matching an intent filter from many apps, *including the system default browser and its Custom Tabs instantiations*, Android will pop up the registered app in a foreground window and instruct it to open that URL.

We also found that some apps define non-https, vendor-specific intents, such as `vnd.youtube` for the YouTube app or `reddit` for the Reddit app. These can be leveraged to execute the app pop-up attack variant when `https` intents cannot be used. The `googlechrome` intent can be used to execute the downgrade attack variant.

[2]We observed that when the default browser does not support Custom Tabs (*e.g.*, the DuckDuckGo and Opera browsers), apps open links using the standalone default browser app instead of opening them in an IAB.

Social Media App Name	Ways to access the IAB
Instagram	Story, Bio, Ads, Messages
Facebook	Comments, Bio, Posts, Messages, Ads
Reddit	Posts, Comments, Ads
Snapchat	Messages, Ads
LinkedIn	Posts, Comments, Bios, Ads
Teams	Messages
Gmail	Emails, Google Chat
Discord	Messages, Server Posts
TikTok	Business Accounts

Table 3: Ways in which apps can open a URL in their IAB.

Resource	Binding method
(privately shared) Youtube video	sharing
(privately shared) Google Drive video	sharing
(privately shared) DropBox video	sharing
(public) LinkedIn video post	blocking

Table 4: Resources usable for the embedding attack variant.

Visiting the Attack Page. Users can be lured to the attack page from apps that have an IAB or from apps that do not have an IAB. Table 3 shows that for ten popular social media apps, there is a multitude of avenues to lure the user to open the attack page in the app's IAB. Apps that do not have an IAB can simply expose a navigable URL to the attack page, which will then be opened in the default browser app.

3.3 Attack Variants

Guided by the multiple ways in which users access resource-sharing services on Android devices, we now identify three attack variants that can be used to execute targeted deanonymization attack on these systems. Next, we explain how the attacker chooses among these variants based on the capabilities of the user's browser, and finally provide examples of some end-to-end attacks.

Embedding Variant. This attack variant is similar in design to the one used in the desktop setting [36]. Here, the attacker places an `<iframe>` HTML tag in the attack website, embedding in it one of the resources shown in Table 4. A JavaScript script also present on the attack page then takes cache measurements while the resource is being loaded and rendered. The main advantage of this attack variant is that it does not require any user interaction once the user is lured to the attacker's webpage. Its main disadvantage with regard to the mobile setting is that it requires the user to be logged into the resource-sharing service through the browser loading the attack page. In addition, the browser must support third-party cookies, so that the authentication cookies for the resource-sharing service are attached to the request for the embedded resource. As we show below, the most commonly used mobile browser, Google Chrome, does support third-party cookies, while others such as Firefox, DuckDuckGo, Edge, Brave, and Tor do not. We finally note that some resource-sharing services ignore cookies from cross-site requests, preventing this form of attack.

App Pop-up Variant. This variant, which is novel to the mobile setting, leverages the existence of dedicated apps for resource-sharing services on mobile devices, as well as the intents feature of Android. The attacker first lures the victim to click anywhere

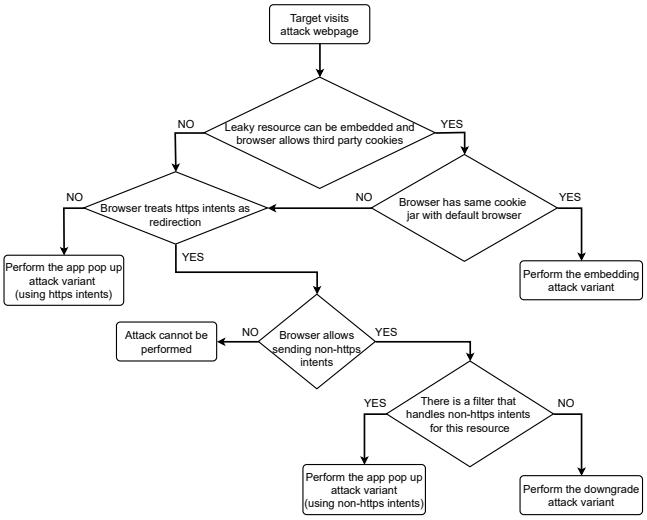


Figure 2: The decision tree for a TD attack on mobiles.

on the attack page, and then uses the standard JavaScript API `window.open(SD-URL)` to open a resource shown in Table 5, either using an `https` intent or a `non-https` vendor-specific intent, as described earlier. This request causes Android to load the shared resource in a foreground window using the dedicated app for that resource (*e.g.*, a YouTube video is opened using the YouTube app). The attack page remains active in the background, taking cache measurements while the resource is being loaded and rendered in the foreground app window. The main advantage of this variant is that it does not require the user to be logged into the resource-sharing service through the browser loading the attack page. Its main disadvantage is its lack of stealth: the app pop-up attack variant requires user interaction, and additionally opens a new window in the foreground on the user's device. As we note in Section 3.6, in the mobile setting this behavior is considered annoying, but not unusual.

Downgrade Variant. There are some scenarios in which neither the embedding nor the app pop-up attack variants can be performed. For example, when using the Tor browser, third-party cookies are not allowed, and the browser treats `https` intentsas redirection. For such scenarios, we propose an approach to "downgrade" the user's browser to the Chrome browser, which is the most permissive with regard to the targeted deanonymization attack. In the downgrade attack variant, the attack page sends an intent using the `googlechrome` scheme to re-open the attack page in the Chrome browser. This can be done upon loading the attack page, by defining an event handler associated with the `onLoad` event, which calls `window.open(googlechrome://attack-page-URL)`. Tor then asks the user to confirm opening the attack page in another app. If the user agrees, the attack page will open in the Chrome browser app, where both the embedding and the app pop-up attack variants can be performed successfully.

3.4 Choosing the Right Attack Variant

We now describe how the attacker chooses among the three attack variants based on the capabilities of the user's browser, following

Resource	Binding method
(privately shared) Youtube video	sharing
(privately shared) Google Drive video	sharing
(privately shared) Dropbox video	sharing
(privately shared) OneDrive video	sharing
(public) LinkedIn video post	blocking
(public) Instagram profile	blocking
(public) Facebook video post	blocking
(public) Reddit video post	blocking

Table 5: Resources usable for the app pop-up attack variant.

the decision tree shown in Fig. 2. The decision tree begins with the attacker's choice of leaky resource, which is pre-determined by the target's public identifier, and concludes with a choice of the most effective attack variant to be used.

Generally, the *embedding attack variant* is the preferred choice for the attacker, as it is the most stealthy — the resource can be embedded using an invisible iframe (*i.e.*, one that has a 0-pixel height and width), and it requires no user interaction. There are, however, several preconditions for choosing this attack variant:

First, the browser loading the attack page must *support third-party cookies*. This support can be tested in real time using well-known techniques [4]. Among the tested browsers, we found that in the default configuration, Chrome and Opera support third-party cookies, whereas other browsers such as Firefox block them.

The second precondition is that the user must be *logged into the resource-sharing service* through the browser loading the attack page. This happens naturally for Google services such as YouTube and Google Drive when the default browser is Chrome, because Chrome comes pre-installed as the default browser on Android devices and Android users are strongly encouraged to "sign into Chrome", effectively causing the browser to log into all Google services. For the other services, this assumption might be less natural.

Finally, the resource-sharing service must *honor cookies from cross-site requests*. Notable services which ignore such cookies are Instagram and Facebook, as studied in more detail by Zaheri et al. [36].

If either of these preconditions is not satisfied, the attacker should next consider the *app pop-up attack variant*. This attack works in more settings, but it is slightly more intrusive, as it requires the user to click once on the attack page, and also opens a new window in the foreground on the user's device. The following preconditions must be satisfied for the app pop-up attack variant to be possible:

First, this variant requires that the target's browser is able to open the shared resource in a dedicated app. For a majority of the attack scenarios, this can be done using `https` intents. However, some browsers, such as the Tor browser or the IABs of Instagram and Facebook, are configured to treat `https` intents as a redirection in the browser to the intent's URL. This will cause the attack to fail, because the browser navigates away from the attack page and cache measurements cannot no longer be taken. Even in these scenarios, we found the app pop-up variant can still be successful by using `non-https` intents, which will lead to opening the leaky resource in its specific app. Specifically, we identified two apps that define filters for such intents: YouTube handles intents sent with `window.open(vnd.youtube:ID)` to open the video identified by `ID`, and Reddit handles intents sent with `window.open(reddit://reddit/PATH)` to open a Reddit resource.

	Embedding variant	App pop-up variant	Downgrade variant
Google Chrome	Yes	Yes	N/A
Firefox	No	Yes	Yes
DuckDuckGo, Edge, Brave	No	Yes	No
Opera	Yes	Yes	No
Tor	No	Yes*	Yes
Instagram IAB, Facebook IAB, Facebook Messenger IAB, TikTok IAB	Yes*	Yes*	No
Reddit IAB	Yes	No	No

Table 6: Attack variant availability in various browsers.

The second precondition is that *the attack page must not be loaded in the IAB of the dedicated app that is responsible for handling the type of leaky resource used for the attack*. For example, assume an attacker wants to use a target's LinkedIn user ID for the attack. If the attack page is loaded in the LinkedIn IAB, then attempting to open a LinkedIn post in a new window will result in the LinkedIn app closing the IAB and opening the post directly in the LinkedIn app. In this scenario, the attack will fail because the attack page can no longer take cache measurements. The attacker can avoid such scenarios by luring users to load the attack page in the IAB of another app different from LinkedIn or in a dedicated browser app.

Finally, we make the natural assumption that the user is logged into the resource-sharing service through the dedicated app for that service installed on their mobile device.

The attacker's final recourse, when neither the embedding nor the app pop-up attack variants are possible, is to use the *downgrade attack variant*. This variant causes the attack page to be re-opened in the Chrome browser, where both the embedding and the app pop-up attack variants can be used, depending on the properties of the chosen leaky resource. Its only requirement is that the browser loading the attack page allows sending `googlechrome` intents. This variant is the least stealthy of the three, and requires user interaction. It relies on luring less experienced users to give consent when presented with a choice to re-open the page in a different browser, especially when the attacker frames the choice accordingly (*e.g.*, claim that some functionality only works in Chrome).

A detailed list of browsers (either standalone or IAB) and their support for the different attack variants is provided in Table 6. For the Chrome, Firefox, Edge, Brave, and Tor browsers, the table refers to either a standalone browser app, or a Custom Tabs instantiation of the browser as an IAB for apps such as Gmail, LinkedIn, Teams, Discord, SnapChat, Slack, Yahoo Finance, Google Discover; for the DuckDuckGo and Opera browsers, the table refers to a standalone browser app. In the Table, the "Embedding variant" column indicates whether the browser allows third-party cookies by default, thus enabling the embedding attack variant. The four specialized IABs (Instagram, Facebook, Facebook Messenger, TikTok) maintain their own cookie jar, distinct from the default browser, and are identified by "Yes*" in the table. The "App pop-up variant" column indicates whether the browser allows sending an `https` intent to open the resource in a new window, thus enabling the app pop-up attack variant. For some browsers, the app pop-up attack variant only works with specific non-https intents; these are identified by "Yes*". The "Downgrade variant" column indicates whether the browser allows sending `googlechrome` intents.

Tables 4 and 5 list the resources which can be used for the embedding and app pop-up variants, respectively. In both variants, the attacker can bind a resource to the target's identity either by sharing it privately with the target (sharing approach), or by blocking the target (blocking approach). For resources that require the blocking-based approach, Appendix B describes what it means to block a user.

3.5 Examples of End-to-End Attacks

We now provide several attack scenarios, which together illustrate the large attack surface for mobile targeted deanonymization.

Scenario 1 (embedding variant, system browser): To target a user based on their Gmail account, the attacker privately shares a YouTube video with the target using the target's Gmail address (which is implicitly a YouTube account). The attacker then embeds the shared YouTube video into an attack page. The target is lured to click on a phishing URL, either in the Chrome browser app or in an app that doesn't have an in-app browser (IAB), such as the Gmail client. In both cases, the target will load the attack page in the default browser app on the Android device, which is usually the Chrome browser. Since the target is logged by default into their Google account in the system Chrome browser, and the Chrome browser allows third-party cookies by default, the embedded YouTube video will be successfully loaded only if the visiting user is the target. Finally, the attacker uses a cache side-channel attack to determine whether the video was loaded successfully, and thus infer whether the visiting user is the target.

Scenario 2 (embedding variant, in-app browser): Let us assume a situation similar to the previous scenario, but this time the user is lured to the attack page through interaction with the LinkedIn app, for example by clicking a link in one of app's sections that exposes clickable URLs, such as "Posts", "Comments", "Bios", or "Ads". In this case, the attack page will be opened in LinkedIn's in-app browser (IAB), which is a custom tabs instantiation of Chrome. Since LinkedIn's IAB inherits the cookies from the Chrome browser, the attack will also succeed.

Scenario 3 (app pop-up variant): Let us assume now that the target user is not identified by their Gmail account, but rather by their Facebook account, and that the user only uses the Facebook app to access their account, without ever logging into Facebook through the system browser. The attacker makes a public Facebook post containing a video, and then blocks the target on Facebook using the target's Facebook ID. The target is then directed to an attack page using the Chrome browser and lured to click anywhere on

the page. Using the Android intent system, the attacker causes the Facebook post to be opened in the foreground using the Facebook app, while the attack page remains in the background, taking cache measurements. Since the Facebook post will be loaded for everyone else except the target (who is blocked), the attacker can detect whether the Facebook post video is playing in the app, leading to a successful attack. This scenario also encompasses users who do not log into their Google account through the system browser, or who use browsers that block third-party cookies by default, such as Firefox, Edge, or Brave. For these, a privately shared YouTube video will be opened in the YouTube app, while the attack page will take cache measurements.

Scenario 4 (downgrade attack variant): The target's setup is the same as in Scenario 1. However, the target loads the attack page in a browser that prevents the embedding and app pop-up attack variants. For example, the target uses the Tor browser when visiting unknown websites or traveling to foreign countries, and loads the attack page in Tor, which does not allow third-party cookies and treats `https` intents as an in-browser redirection. Or, the target loads the attack page in a specialized IAB, such as the Instagram IAB, which also treats `https` intents as an in in-browser redirection. In these cases, the attack page tries to "downgrade" the target's browser to Chrome by using non-https Android intents such as the `googlechrome` intent to instead re-open the attack page in the Chrome browser, and then proceeds as in Scenario 1.

3.6 Discussion

The Issue of Stealth. The app pop-up and downgrade attack variants presented above are far more intrusive than the attacks evaluated in the desktop setting: they require user interaction, pop up a window that covers the target website, and require additional interaction to close. We argue that these attack variants are still realistic in the mobile setting, despite their reduced stealth: In a mobile environment, it is common for apps to open other apps or to display ads in a new window; thus, a website that opens full-screen windows in the foreground is not necessarily suspicious. To induce the target to click on the attack page, the attack page can display what looks like an advertisement that covers a significant portion of the page and remains persistent on the screen even when the user scrolls down, thus hindering the user's ability to view the contents of the page. The fake ad contains what looks like an "X" button and the user is likely to click on it to close the ad and view the page they were originally trying to visit. If a new window opens as a result of that click, the user may not even view it as suspicious, as they may assume they didn't correctly click on the close button. As another example, the attack page displays a fake button with a convincing text to make the user click on the area of that fake button. Suspicions can also be explicitly reduced by making the user aware that a new app will be opened (*e.g.*, "to continue, click and a video will be shown about how to use the service").

In some IABs, when the app pop-up attack is executed using non-https intents, the user is presented with a dialog window informing them that the website is attempting to open an external app and asking them to confirm if they want to continue. Some users may find this dialog suspicious and may not continue. This can again be remediated by providing the user with relevant guidance. We finally note that the attack often concludes in less than a second, as we show in our evaluation section. Thus, it may not even be relevant whether the victim becomes suspicious.

Specialized IABs. The four specialized IABs (we studied Instagram, Facebook, Facebook Messenger, TikTok) allow sending third-party cookies, but maintain their own cookie jar, distinct from the default browser. As such, the embedding attack variant is possible in these IABs under two scenarios: 1) the user has previously logged into a resource sharing service in the IAB, or 2) the user is in the IAB and uses her Google or LinkedIn credentials to log into a website. In the second scenario, the attack leverages the IAB vulnerability described in Appendix A. In all other settings the embedding attack will not work on these IABs, requiring the attacker to select another variant. To note the limited reach of the embedding attack variant on these IABs, we mark them with a "Yes*" in Table 6.

4 Evaluation

4.1 Experimental Setup

Most of the attacks described in this section were performed on a Google Pixel 6a phone running Android 16, with Google Tensor chipset, 6GB RAM memory and 128GB storage. In Section 4.2.3, we reproduce a subset of the results on another phone model (Samsung Galaxy S25 FE), showing that the attack is not specific to a particular phone model or manufacturer. To run the attack and collect the cache measurements, we used an automation framework based on Selenium [3] and Appium [1] with the UIAutomator2 driver running on a testing laptop. Selenium is a tool used to automate browser interactions based on automation scripts written in Python. Selenium interacts with Appium (a node.js application designed to facilitate UI automation), which parses and applies the automation scripts to the Android phone. Appium interacts with the phone through the Android Development Bridge (ADB). ADB will interpret commands received and execute them on the phone; it will also read data from the phone screen and relay it back to Appium. In our experiments, the phone was connected to the testing laptop via a USB-C wire, but even a wireless connection to ADB will work. The attack page leverages a leaky resource, corresponding to one of the attack variants described in Sec. 3. We include the versions used for browsers and apps in Appendix C.

Attack Methodology. The attack has two phases, a training phase and an online phase. In the *training phase*, the attacker collects cache traces that correspond to loading and not loading a leaky resource, and uses them to train a machine learning classifier to detect the cache signature that corresponds to successfully loading the leaky resource. The attacker can run the training phase to build machine learning models for a variety of combinations of leaky resource, browser, and device hardware. The training phase happens in advance, without involving the potential victims.

In the *online phase*, the attack page takes cache measurements while the leaky resource is loaded and rendered by a user. These measurements are then uploaded to the attacker's server, where they are passed through the trained classifier to determine whether the user is the target. The attack accuracy depends on the *attack duration*, which is the amount of time for which the attacker needs

Resource	Attack accuracy	Attack duration	Attack variant
YouTube video	95%±4.47%	1s	embedding
Google Drive video	100%±0%	1s	embedding
DropBox video	97%±4%	1s	embedding
LinkedIn video post	99.5%±1.5%	1s	embedding
YouTube Video	92.5%±4.61%	1s	app pop-up
Google Drive video	97.5%±3.35%	1s	app pop-up
DropBox video	93.4%±5.83%	1s	app pop-up
OneDrive video	97.5%±3.35%	3s	app pop-up
LinkedIn video post	97.0%±2.45%	2s	app pop-up
Instagram profile	96%±3.74%	3s	app pop-up
Facebook video post	96.0%±3.0%	1s	app pop-up
Reddit video post	92.5%±3.35%	2s	app pop-up

Table 7: Experimental scenarios with different resources and attack variant combinations. In all scenarios, the attack page was loaded in Google Chrome.

to collect cache traces in the online phase. For all tested attack scenarios, the attack duration is under 3 seconds.

To take cache measurements, we use JavaScript code based on the PP0 repository [22]. For most attack scenarios, we use the cache occupancy method; for a small number of scenarios, such as those involving the Tor browser which provides a coarse timer resolution, we use the sweep counting method, as described in Sec. 2.1.

Data Analysis Methodology. We follow the data analysis methodology used by Zaheri et al. [36]. Specifically, we use supervised machine learning to analyze the cache measurements. The classifier used is logistic regression with 1000 max iterations. For each attack setting, we collect a dataset of 200 samples (100 for the target and 100 for a non-target user). We train a classifier using a subset of the collected samples, and then use the classifier to predict whether the user loading the attack page is the target.

4.2 Experimental Results

4.2.1 Impact of Leaky Resource. We first tried to answer the question *"Do different leaky resources impact differently the TD attack effectiveness (defined by attack accuracy and attack duration)?"*. Since some of the leaky resources could be used for both the embedding and the app pop-up attack variants, we also tried to understand whether the attack variant impacts differently the attack effectiveness. For all scenarios in this experiment, the attack page was loaded in the Chrome browser, but we varied the resource being leveraged for the attack and we also varied the attack variant whenever possible. Table 7 shows the attack effectiveness for different resource and attack variant combinations.

We observe that the attack accuracy is above 90% for all considered scenarios. In addition, the attack duration is under 1s for a large majority of the considered scenarios. We notice that the resource type does not significantly affect the attack duration, as all the scenarios using the embedding variant are successfully performed under 1s. For the app pop-up variant, the attack duration seems to depend on the time to launch the respective dedicated app, which increases the attack time up to 3s for some of the apps.

4.2.2 Impact of Browser. We now turn our attention to answering the question *"Do different standalone mobile browsers or IABs offer increased resilience to the TD attack?"*. For the scenarios in this experiment, we used the same resource (a privately shared YouTube video), but we varied the browser in which the attack page was loaded. Table 8 shows the attack effectiveness for different browser and attack variant combinations.

Browser app	Attack accuracy	Attack duration	Attack variant
Chrome browser	95%±4.47%	1s	embedding
Opera browser	100%±0%	1s	embedding
Firefox browser	90.5%±6.5%	1s	app pop-up
DuckDuckGo browser	95%±6.71%	3s	app pop-up
Edge browser	89.5%±6.87%	3s	app pop-up
Brave browser	93.5%±5.02%	2s	app pop-up
Tor browser	94.5%±6.1%	3s	app pop-up
Chrome as CT IAB	99.5%±1.5%	1s	embedding
Firefox as CT IAB	96%±4.36%	2s	app pop-up
Edge as CT IAB	90.5%±5.22%	2s	app pop-up
Brave as CT IAB	97.5%±2.5%	2s	app pop-up
Tor as CT IAB	96.5%±2.29%	3s	app pop-up
Instagram IAB	91.5%±3.2%	2s	app pop-up
Facebook IAB	90%±5.47%	2s	app pop-up
Facebook Mssngr IAB	93.5%±5.02%	2s	app pop-up
TikTok IAB	98%±3.32%	1s	app pop-up
Reddit IAB	99.5%±1.5%	1s	embedding
Reddit IAB	99.5%±1.5%	1s	app pop-up

Table 8: Experimental scenarios with different browsers and attack variant combinations. In all scenarios, the resource used for the attack was a privately shared YouTube video.

These results show that the choice of browser, including the highly secure Tor browser, custom tabs instantiations of various browsers and specialized IABs, does not provide additional protection. Otherwise, the embedding attack variant remains effective at under 1s, whereas the app pop-up variant is comparable to the earlier scenarios that use the Chrome browser app (in which the increased attack duration is attributed to the time needed to open the dedicated app).

4.2.3 Reproduction on Additional Phone Model. To show that our attack is not specific to a particular phone model or manufacturer, we reproduced a subset of the results on a Samsung Galaxy S25 FE phone, running Android 16, with Exynos 2400 chipset, 8GB RAM memory and 128GB storage. Table 9 shows the attack accuracy for several scenarios selected from the previous experiments, each representing a different attack variant: The *embedding attack* variant was evaluated using a YouTube video and reproduced both in the standalone Chrome browser app, and in a Custom Tabs in-app browser hosted inside the LinkedIn app. The *app pop-up* variant was evaluated using a YouTube video and a LinkedIn video post, and reproduced both in the standalone Chrome and Firefox browser apps. Finally, the *downgrade* attack variant was for the Tor browser using `googlechrome` intents. The results in Table 9 confirm that the attack reproduces on the Samsung Galaxy device, with attack accuracies comparable to the Pixel device for most scenarios.

5 Defenses

Several defenses have been proposed to counter or mitigate targeted deanonymization (TD) attacks in the desktop setting. In this section,

Browser app/ shared resource	Attack accuracy	Attack duration	Attack variant
Chrome browser/YTV	85%±5.48%	1s	embedding
Chrome as CT IAB/LIVP	74.5%±6.50%	1s	embedding
Chrome browser/YTV	96%±3.74%	2s	app pop-up
Chrome browser/LIVP	99.5%±1.50%	2s	app pop-up
Firefox browser/YTV	98%±2.45%	1s	app pop-up
Firefox browser/LIVP	94%±7.68%	1s	app pop-up
Tor browser/`googlechrome`	verified		downgrade

Table 9: Experimental scenarios reproduced on the Samsung Galaxy phone, using either a YouTube video (YTV) or a LinkedIn video post (LIVP) as a shared resource.

we review the main differences in the attack surface of targeted deanonymization attacks between the mobile and desktop settings, and discuss the effects of these differences on the efficacy of existing defenses. We then propose a defense strategy tailored for the mobile environment. We note that since the targeted deanonymization attack targets the browser's rendering process, and not the network stack, defenses against the attack must be deployed in the application layer. Thus, network-level defenses such as the use of a VPN service will be ineffective, both in the desktop and in the mobile environment.

5.1 Challenges in Existing Approaches

One of the most significant differences between the desktop and mobile environments is the decentralization of ways in which a user can access a resource: On a desktop, users only access resource-sharing sites through their browser. On the mobile phone, however, in addition to using the system's main browser, users can also access the resource-sharing site through its own proprietary app, and additionally through any other app which offers an in-app browsing feature. An outcome of this difference is the limited effectiveness of "incognito mode" in the mobile environment — in the desktop environment, as soon as the user enters incognito mode, any websites opened by the user, whether arriving by instant message, email, or any other origin, are opened without access to the user's standard cookie jar. In the mobile environment, in contrast, incognito mode does not apply to the resource-sharing site's proprietary app, which has its own authentication token independent of the browser's cookie jar, nor does it apply to in-app browsers, which always have access to the user's standard cookie jar, even if the user has an incognito session open at the same time.

As a result, in the mobile setting, incognito mode will protect only against embedding attacks on the system's main browser. Embedding attacks against any in-app browser, app pop-up attacks against any browser, and downgrade attacks, will still be vulnerable to targeted deanonymization. This situation also limits the effectiveness of protections deployed as browser extensions: Zaheri *et al.* [36] proposed such a defense, which detects if a webpage is retrieved differently when the HTML request contains or does not contain cookies, and offers the user a choice to reload the page with cookies if they trust the page. Unfortunately, the most commonly-used browser available for Android devices, Google Chrome, does not currently support browser extensions. Even if the user installs an alternative browser which does support extensions, such as Firefox, the use of browser extensions would be challenging in an IAB. In addition, the app pop-up attack variant relies on the user's authentication status in the dedicated app, bypassing any browser-level protections.

Another approach explored in the desktop setting is the injection of noise into the CPU cache, to render cache readings useless. Basak *et al.* [7] proposed such a defense, which was also implemented as a browser extension targeting a desktop environment. Such a defense could potentially be effective for mobile devices, when implemented as a standalone app that runs in the background. However, an approach based on injecting noise was found to impose a significant overhead in the desktop setting, and is likely to be impractical in a mobile setting, which has additional processing power and battery constraints compared to a desktop environment.

5.2 Toward a Mobile-Centric TD Defense

An effective defense against targeted deanonymization must deal with all of the challenges described above. First, to deal with decentralization, both the main browser and the in-app browsers should refrain from passing links to external apps during privacy-sensitive situations. This defense seems possible to apply without fundamental changes to the Android mobile operating system. In fact, users can simulate this behavior by manually editing the user settings that define which apps should be used to open certain links. At installation, an Android app can declare intent filters indicating that the app should be launched to handle a subset of URIs that contain specific network domains. Users can then disable this behavior by manually entering the settings page for each app and unchecking its "Open by Default" options.

This defense, however, only applies to `https` intent filters. For known non-https intents that could be leveraged to execute the pop up attack variant, such as YouTube and Reddit intents, an effective defense is to disable the respective app. Users could deploy this defense temporarily when faced with a potentially high-risk situation, such as visiting an untrustworthy website or traveling to certain regions, and then re-enable the app at a later time. Disabling and re-enabling an app is straightforward, as an option available under the "App info" screen. A more extreme option is to uninstall the app completely, but this is not always possible. For example, the YouTube app, which can leveraged to execute the pop up attack variant using YouTube intents is installed by default on all Google Android phones and cannot be uninstalled.

Another approach, which includes a slightly more elaborate change to the architecture of the Android operating system, is to define incognito mode as a *device-wide state* that applies to all apps, and not only to the main browser. When incognito mode is enabled, all apps, including the main browser, IABs, and dedicated apps, would refrain from using the user's standard cookie jar (or its app-specific equivalent) to authenticate to remote servers. A simplified version of this approach would be to allow apps to control the incognito status and intent filter list of any IAB they create via an extended API. An app can open links in its IAB instead of opening them in the respective Android app, by controlling the `setSendToExternalDefaultHandlerEnabled` option when creating a Custom Tabs intent [2]. This prevents the pop up attack variant using `https` intents, greatly reducing the attack surface. As a more extreme approach, an app's IAB could choose to block certain non-https intents that can be leveraged to conduct the attack.

For example, the IABs of Instagram, Facebook, Facebook Messenger, TikTok and Reddit block `googlechrome` intents. Using these settings, applications that deal with privacy-sensitive content, such as finance or health apps, could create IABs that always operate in incognito mode and never open links in external apps.

Finally, another possible security measure would to restrict the ability of background processes to take fine-grained time measurements, or even disable JavaScript for background processes. In contrast to the desktop setting, where users often have multiple windows open simultaneously, mobile users typically interact with a single app at a time. Thus, restricting background processes could be a viable defense which could deny the attacker page the ability to collect useful data, while still preserving usability.

5.3 Designing TD-Resistant Websites

Resource-sharing services can take steps towards making their resources less "leaky". Because the attack relies on observing the impact of loading a resource on the CPU cache, loading a resource on failure can mitigate the predictive power of the attack. In other words, resource-sharing services could return a result that does not differ significantly in resources to display for both cases when a resource can and cannot be displayed. This strategy is already employed by TikTok which, upon failing to load a resource, loads the user's "for you page" that also contains videos and thus degrades the usefulness of cache side channel. Including this failure behavior would defend against both embedding and pop up attacks.

6 Related Work

6.1 Side channel-based attacks

CPU cache-based attacks on mobile devices have been conducted as early as 2016 [20], such as inferring keystrokes and swipe actions on the touchscreen, and attacking cryptographic implementations. Most mobile devices use ARM CPUs, which have a different cache organization and instruction set than Intel x86 CPUs. Thus, the main challenge was how to adapt techniques known to be effective for cache attacks from the desktop to the mobile setting. Gulmezoglu et al. [13] show how an unprivileged, zero-permission app can use the last-level cache of ARM processors to profile activity on Android phones, such as detecting running applications, opened websites, and streaming videos. To infer such information, they leverage deep learning techniques that can overcome the inherent noisiness of cache monitoring. Cronin *et al.* [10] propose an optimized cache occupancy channel for ARM processors based on a System-on-a-Chip design that uses the system-level cache to perform website fingerprinting. We focus on executing targeted deanonymization attacks in the browser, using JavaScript, as opposed to assuming that users will install a malicious app. Also, our goal is to infer the identity of a target simply based on their public identifier.

More recently, **GPU-based side channels** have also been used to infer private information on mobiles. Karimi *et al.* [14] introduce a timing side channel that exploits the GPU cache locality by correlating the execution time with the number of unique memory requests to recover an AES-128 encryption key. Cronin *et al.* [10] introduce a GPU contention channel using the GPU.js JavaScript library to perform website fingerprinting. In doing so, they overcome several challenges stemming from architectural details of GPUs on various mobile devices, including the lack of a timer to measure the execution of GPU computational kernels. Yang *et al.* [34] exploit GPU performance counters to infer the amount of screen display changes at the granularity of individual pixels. This allows an unprivileged attacking app to eavesdrop the user's credentials when typed on an on-screen keyboard. In the web context, Laor *et al.* [17] showed how WebGL can be used to perform individual device fingerprinting, by measuring the performance of individual execution units in the GPU. Wu *et al.* [33] introduce a rendering contention side channel that measures the time needed for a mobile browser to render a sequence of frames. This channel is then used to perform history sniffing, website fingerprinting, and keystroke logging attacks. Our work focuses on exploiting CPU cache side channels, but GPU side channels could be used as an alternative mechanism to conduct targeted deanonymization on mobiles.

Other types of side channels exist that can be used to violate user privacy on mobile phones. For example, the return values of system calls can be collected over time and analyzed using deep learning models to infer app launching and website browsing [8], and even more fine-grained user behavior such as tapping, swiping, scrolling, and other on-screen gestures [30]. Yan *et al.* [19] use undocumented Android features to break Android's app sandboxing and leak private information between apps. A malicious unprivileged app can use a vulnerability in the dynamic code loading mechanism using package context (DICI) to monitor runtime behavior of other apps via a cache-based side channel and infer driving routes taken by a navigation app and keystroke dynamics of a communication app. However, this approach can be prevented by fixing the vulnerability and is not applicable to apps that do not use DICI. Taneja *et al.* [29] use CPU power, frequency and temperature measurements from internal sensors to perform browser-based pixel stealing and history sniffing attacks on ARM System-on-a-Chip mobile devices.

6.2 Privacy threats from in-app browsers

Several large-scale empirical studies reveal security and privacy violations when Android apps interact with web content. Kuchhal *et al.* [16] examine if Android apps use WebViews and CustomTabs in a manner that aligns with user security and privacy considerations. They find several apps are misusing WebViews to show arbitrary web content within the app while monitoring and modifying the web content behavior without explicit consent from the users. Weerasekara *et al.* [32] examine how app developers configure WebViews to bypass default Android privacy protections and facilitate user tracking through JavaScript code. These findings are aligned with our findings that in-app browsing exposes a large surface for privacy attacks.

6.3 Defenses against side channel-based attacks

In general, defending against side channel attacks is difficult, because such attacks take advantage of hardware-level properties of the victim's device and therefore bypass software-imposed boundaries. On desktops, there are two categories of defense approaches against side channel attacks. The first tries to make the attack impractical by reducing the signal-to-noise ratio of the side-channel trace. The second tries to make the attack theoretically impossible by removing all dependencies between the side-channel trace

and any secret-bearing computation. In the first category, noise injection was shown to be an effective defense [7, 18, 24] against website fingerprinting attacks, even when faced with a strong adversary that trains a machine learning model in the presence of noise. However, this approach can introduce a significant performance overhead. More so, it is unlikely that noise-based defenses would be feasible on mobiles, which have additional computational and power limitations compared to their desktop counterparts.

Cho *et al.* [9] propose a software-based cache side channel mitigation for ARM devices that ensures each process has a private space to safely access private data. However, this approach requires to identify, annotate, and recompile the sensitive code that needs to be protected, which is not always feasible.

7 Conclusion

In this paper, we have seen that targeted deanonymization (TD) attacks represent a significant threat to mobile devices. An attacker has a plethora of avenues to conduct the TD attack, covering multiple mobile browsers, in-app browsers of multiple popular social media apps, and multiple popular resource-sharing services. We found that the attack surface even includes attack variants that are not available in the desktop setting. Thus, an overwhelming majority of mobile users are vulnerable to such attacks. The attacks are straightforward to deploy and run under 3 seconds, with many of the scenarios even running under 1 second.

We have also discussed several defenses, ranging from ones that can be enabled right away by users, alas at the cost of sacrificing some functionality, to defenses that could be deployed by the OS and browser vendors, or by the providers of resource-sharing services. It remains an open challenge to build a defense that is both practical to deploy and that preserves current functionality. Ultimately, this is made difficult by the fact that cache side channels take advantage of hardware-level properties of the victim's device and therefore bypass software-imposed boundaries.

Our work has focused on Android devices and has demonstrated end-to-end attacks targeting a single user. Future work includes efficiently scaling the attack to target a large group of users and exploring attack feasibility as well as defense options on iOS devices.

References

[1] [n. d.]. Appium. https://appium.io/.
[2] [n. d.]. Open links in Android apps. https://developer.chrome.com/docs/android/custom-tabs/guide-warmup-prefetch#open_webpages_in_native_apps.
[3] [n. d.]. Sellenium. https://www.selenium.dev/.
[4] [n. d.]. Third party cookie check for browsers. https://github.com/mindmup/3rdpartycookiecheck.
[5] 2022. XSLeaks Summit 2022. https://tinyurl.com/xsleakssummit2022.
[6] 2023. XSLeaks Summit 2023. https://tinyurl.com/xsleakssummit2023.
[7] Tapan Basak, Reza Curtmola, Hai Phan, and Yossi Oren. 2025. FlipStress: Noise Injection Defenses Against CPU-cache-based Web Attacks. In *Proc. of The 21st EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM '25)*. EAI.
[8] Valerio Brussani. 2022. ASVAAN: Semi-automatic side-channel analysis of Android NDK. arXiv:2204.05911.
[9] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2020. SmokeBomb: effective mitigation against cache side-channel attacks on the ARM architecture. In *Proc. of the 18th Intern. Conf. on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 107–120.
[10] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An Exploration of ARM System-Level Cache and GPU Side Channels. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*. ACM, 784–795.
[11] MDN Web Docs. 2025. Cross-Origin-Opener-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.
[12] MDN Web Docs. 2025. Cross-Origin Resource Policy (CORP). https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).
[13] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining User Privacy on Mobile Devices Using AI. In *AsiaCCS*. ACM, 214–227.
[14] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2018. A Timing Side-Channel Attack on a Mobile GPU. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*.
[15] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *ACM CCS*. 1771–1788.
[16] Dhruv Kuchhal, Karthik Ramakrishnan, and Frank Li. 2024. Whatcha Lookin' At: Investigating Third-Party Web Content in Popular Android Apps. In *Proceedings of the 2024 ACM on Internet Measurement Conference (IMC)*. ACM, 114–129.
[17] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. 2022. DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting. In *NDSS*.
[18] Haipeng Li, Nan Niu, and Boyang Wang. 2022. Cache Shaping: An Effective Defense Against Cache-Based Website Fingerprinting. In *CODASPY 2022 - Proceedings of the 12th ACM Conference on Data and Application Security and Privacy*.
[19] Yan Lin, Joshua Wong, Xiang Li, Haoyu Ma, and Debin Gao. 2024. Peep with a mirror: breaking the integrity of android app sandboxing via unprivileged cache side channel. In *Proc. of the 33rd USENIX Conference on Security Symposium*.
[20] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: cache attacks on mobile devices. In *Proceedings of the 25th USENIX Conference on Security Symposium*. 549–564.
[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*. 605–622.
[22] Yossi Oren. 2021. PP0 GitHub Repository. https://github.com/Yossioren/pp0.
[23] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA (LNCS, Vol. 3860)*. Springer, 1–20.
[24] Son Seonghun, Dipta Debopriya Roy, and Gulmezoglu Berk. 2023. DefWeb: Defending User Privacy against Cache-based Website Fingerprinting Attacks with Intelligent Noise Injection. In *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*. ACM, 379–393.
[25] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*.
[26] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*.
[27] Cristian-Alexandru Staicu and Michael Pradel. 2019. Leaky Images: Targeted Privacy Attacks in the Web. In *USENIX Security Symposium*. 923–939.
[28] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*.
[29] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. 2023. Hot pixels: frequency, power, and temperature attacks on GPUs and arm SoCs. In *Proc. of the 32nd USENIX Conference on Security Symposium*.
[30] Quancheng Wang, Ming Tang, and Jianming Fu. 2024. EavesDroid: Eavesdropping User Behaviors via OS Side Channels on Smartphones. *IEEE Internet of Things Journal* 11, 3 (2024).
[31] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. 2018. User Blocking Considered Harmful? An Attacker-Controllable Side Channel to Identify Social Accounts. In *EuroS&P*. IEEE.
[32] Nipuna Weerasekara, José Miguel Moreno, Srdjan Matic, Joel Reardon, Juan Tapiador, and Narseo Vallina-Rodriguez. 2025. Tracking Without Borders: Studying the Role of WebViews in Bridging Mobile and Web Tracking. In *Proceedings on Privacy Enhancing Technologies (PETS)*.
[33] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. 2022. Rendering Contention Channel Made Practical in Web Browsers. In *31st USENIX Security Symposium (USENIX Security 22)*. 3183–3199.
[34] Boyuan Yang, Ruirong Chen, Kai Huang, Jun Yang, and Wei Gao. 2022. Eavesdropping user credentials via GPU side channels on smartphones. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 285–299.
[35] Mojtaba Zaheri and Reza Curtmola. 2021. Leakuidator: Leaky resource attacks and countermeasures. In *Proc. of the 7th EAI SecureComm*.
[36] Mojtaba Zaheri, Yossi Oren, and Reza Curtmola. 2022. Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses. In *Proc. of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association, 1505–1523.

A Specialized IAB vulnerability to enable embedding attack

Specialized IABs such as the Instagram, Facebook and Facebook Messenger IABs allow sending third-party cookies, but maintain their own cookie jar, distinct from the default browser. As such, the embedding attack variant should normally be possible in these IABs only if the user has previously logged into a resource sharing service in the IAB. However, we found a vulnerability in these IABs that opens additional avenues to execute the embedding attack variant in the IABs.

Identity providers are systems that verify user identities and allow them to sign on to other services. For example, users can use their Google credentials to sign on to a third-party website. When this option is selected, the user is redirected to Google, where the user authenticates using their Google credentials, and then is allowed to sign on to the third-party website based on an assertion made by Google confirming the user's identity. We found that when a user is browsing in a specialized IAB and uses their Google credentials to "Sign in with Google" to a third-party website, the user is authenticated to both the third-party website and to Google, and remains authenticated for the remainder of their IAB session. This is an unintended side-effect, since the user's original intention was to log into the third-party website only. As such, the embedding variant of the TD attack becomes now possible in the IAB, for example using a YouTube or a Google Drive video. The requirement for this is that the identity provider must also host resources that can be leveraged for the embedding attack variant We identified two such providers, Google and LinkedIn, and experimentally validated that this vulnerability exists for the specialized IABs we considered (Instagram IAB, Facebook IAB, and Facebook Messenger IAB). We have performed responsible disclosure regarding this observed behavior.

B Blocking behavior for various services

For the blocking-based approach of the TD attack, the attack page attempts to open a resource that the target cannot access because they are blocked. In the app pop-up attack variant, assuming that user A has blocked user B, when B tries to open one of A's posts, the blocking is handled differently depending on the service:

- Facebook app: B gets her standard Facebook feed instead of A's post.
- LinkedIn app: B gets a screen with an error message stating that the post is not available. In general, B cannot see anything about A's profile, bio or posts.
- Reddit app: B gets a screen with an image with an error message stating that something went wrong.
- Instagram app: B gets a barebones version A's profile, without any pictures for A's posts.

C Software Version Details

The following software versions were used on the Pixel device: Android: version 16; Chrome browser: 142.0.7444.138; Firefox browser: 144.0.2; Brave browser: 1.84.136; DuckDuckGo browser: 5.256.0; Edge browser: 142.0.3595.93; Opera browser: 43.0.4906.86186; Tor browser: 15.0 (140.4.0esr); Instagram app: 406.0.0.58.159; Facebook app: 538.0.0.53.70; Facebook Messenger app: 533.0.0.47.109; TikTok app: 42.6.4; Reddit app: 2025.44.0; YouTube app: 20.46.37; DropBox app: 450.2.2; LinkedIn app: 4.1.1138.1; Google Drive app: 2.25.450.0.-all.aldpi; OneDrive app: 7.43.

The following software versions were used on the Samsung Galaxy device: Android: version 16; Chrome browser: 143.0.7499.109; Firefox browser: 146.0; YouTube app: 20.51.34; LinkedIn app: 4.1.1150.