Pixnapping: Bringing Pixel Stealing out of the Stone Age

Alan Wang
University of California, Berkeley

Pranav Gopalkrishnan
University of Washington

Yingchen Wang
University of California, Berkeley

Christopher W. Fletcher
University of California, Berkeley

Hovav Shacham
University of California, San Diego

David Kohlbrenner
University of Washington

Riccardo Paccagnella
Carnegie Mellon University

Abstract

Pixel stealing attacks enable malicious websites to leak sensitive content displayed in victim websites. The idea, introduced by Stone in 2013, is to embed victim websites in iframes and use SVG filters to compute on, and create side channels as a function of, those websites' pixels. Fortunately, despite the danger, pixel stealing attacks are all but mitigated today thanks to websites and web browsers heavily restricting iframes and cross-origin cookie sharing.

This paper introduces a pixel stealing framework targeting Android devices that bypasses all browser mitigations and can even steal secrets from non-browser apps. Our key observation is that Android APIs enable an attacker to create an analog to Stone-style attacks outside of the browser. Specifically, a malicious app can force victim pixels into the rendering pipeline via Android *intents* and compute on those victim pixels using a stack of semi-transparent Android *activities*. Crucially, our framework enables stealing secrets only stored locally (e.g., 2FA codes and Google Maps Timeline), which have never before been in reach of pixel stealing attacks.

We instantiate our pixel stealing framework on Google and Samsung phones—which differ in both hardware and graphical software. On the Google phones, we additionally provide evidence that the pixel color-dependent timing measured in our attack is due to GPU graphical data compression. We demonstrate end-to-end attacks that steal pixels from both browser and non-browser victims, including Google Accounts, Gmail, Perplexity AI, Signal, Venmo, Google Messages, and Google Maps. Finally, we demonstrate an end-to-end attack capable of stealthily stealing security-critical and *ephemeral* 2FA codes from Google Authenticator in under 30 seconds.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**; **Mobile platform security**.

Keywords

Pixel stealing attacks; Side-channel attacks; Android security

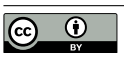
1 Introduction

Pixel stealing attacks have been a significant threat to web security, enabling malicious websites to steal sensitive user data (emails, credentials, etc.) displayed in victim websites. These attacks, starting with Stone's work in 2013 [48], are based on the following idea: while attacker code cannot directly read a victim website's pixels, SVG filters in browsers enable the attacker to perform image transformations on victim pixels embedded in an attacker site's iframes. Stone observed that some of these transformations have pixel-color dependent execution time—enabling a timing side channel that can be used to leak attacker-chosen victim pixel values. Subsequent years witnessed significant follow-on work, exploring different side channels that arise due to SVG filters [21, 28, 34, 42, 43, 49, 51, 53].

Fortunately, Stone-style pixel stealing attacks are all but mitigated by major browsers and websites today. Specifically, the `X-Frame-Options` header and the `frame-ancestors` content-security-policy directive allow sensitive websites to prevent being framed by other websites. Additionally, by default, modern web browsers restrict cookies from being sent with cross-origin requests, preventing secrets from appearing in an attacker site's iframes. While some browsers allow websites to override these settings, most websites do not. As a result, Stone-style pixel stealing is no longer considered a threat [16]. This is indirectly confirmed by several recent Stone-style pixel stealing attacks, which explore more downstream side channels and remain unmitigated [1, 3–5]. One is left to ask: do pixel stealing attacks have a future as a relevant security threat?

In this paper, we introduce a pixel stealing framework targeting Android devices that bypasses all browser mitigations and is additionally able to leak pixel values from non-browser apps. Our key observation is that Android APIs enable malicious apps to construct an analog to Stone-style pixel stealing attacks *outside of the browser*. Recall that Stone-style attacks send victim pixels to the rendering pipeline by opening a victim website in an iframe and compute on those pixels by applying a stack of SVG filters on top of the iframe. Our attack sends victim pixels to the rendering pipeline by opening a victim activity using *intents* and computes on those pixels using *a stack of semi-transparent activities* positioned in front of the victim activity. Similarly to the aftermath of Stone's work, we anticipate there will be a number of side channels able to exploit pixel color-dependent behavior in the stack of activities. In fact, Android provides far richer facilities (e.g., the attacker can run native code) to measure these side channels relative to browser-based Stone-style attacks. We exploit one side channel, due to GPU graphical data compression [51], for the attacks in this paper.

Intents enable our attack to reach a broader swath of victim pixels than was possible using iframes. While iframes can only be used to render web pages, intents can be used to access pixels in both web pages and native Android apps. These include secrets that are only stored securely on the victim device—such as SMS and



Signal messages, 2FA codes, and the Google Maps Timeline—which have *never before been in reach of pixel stealing attacks.*

Stacks of activities enable similar functionality as Stone's stack of SVG filters but require some effort to exploit. Consider: SVG filters in Stone-style attacks are quite powerful—enabling an attacker to perform arbitrary convolutions, scales, blurs, etc. By contrast, the only comparable transformation available through Android activities is *blur*. Despite this limitation, we show how it is still possible to compute on attacker-chosen victim pixels in a fashion that creates measurable pixel color-dependent timing disturbances.

We instantiate our framework on four Google Pixel phones and a Samsung Galaxy S25 phone. For the Google phones, we perform a root cause analysis of pixel color-dependent timing differences and show that the root of the side channel is likely GPU graphical data compression [51]. Relative to the Google Pixel phones, the Samsung Galaxy features quite different hardware and graphical software. Our goal in evaluating both is to demonstrate how our framework can be applied to a variety of Android devices.

We demonstrate end-to-end attacks capable of stealing security-critical and *ephemeral* secrets while hiding the attack from the user. Inside the browser, we steal pixels from a number of websites (Google Accounts, Gmail, Perplexity AI), all of which are protected against Stone-style pixel stealing attacks. Outside the browser, we steal pixels from Google Maps, Google Messages, Venmo, Signal, and Google Authenticator. Several of these apps contain secrets that are fundamentally out of reach for Stone-style pixel stealing attacks. Our attack on Google Authenticator uses additional optimizations—including the OCR-style technique proposed in Stone's original paper [48]—to leak 2FA codes before they disappear.

This paper's contributions can be summarized as follows:

(1) We propose a pixel stealing framework in Android which enables attackers to reach *any* website and *any* app.
(2) We demonstrate our framework's applicability to multiple Android devices by instantiating it on four Google Pixel phones (6, 7, 8, and 9) and a Samsung Galaxy S25 phone, which differ in both hardware and graphics software.
(3) We demonstrate several end-to-end attacks that steal sensitive pixels from both browser and non-browser apps while hiding themselves from the user. Notably, these include an optimized end-to-end attack that recovers ephemeral 2FA codes from Google Authenticator in under 30 seconds.

Disclosure. We disclosed our findings to Google on February 24, 2025. Google rated the issue as "High severity", assigned CVE-2025-48561, and committed to awarding us a bug bounty. On September 2, Google released a patch for our attacks, which we became aware of on September 4. We then discovered a workaround to the patch and additionally observed that it does not mitigate Section 4.2's instantiation. On September 8, we disclosed these new findings to Google, which rated them as "High severity". On September 19, we also disclosed to Samsung that Google's patch was insufficient to protect Samsung devices. As of October 13, we are still coordinating with Google and Samsung regarding disclosure timelines and mitigations. Separately, we also disclosed Section 3.1's app list bypass vulnerability to Google on April 23, 2025; Google rated it "Low severity" and resolved the report as "Won't fix (Infeasible)".

```
1  Intent intent = new Intent(Intent.ACTION_VIEW,
       Uri.parse("https://www.google.com"));
2  intent.setPackage("com.android.chrome");
3  startActivity(intent);
```

Listing 1: Example of an implicit intent that opens the Chrome browser to `www.google.com`.

2 Background

2.1 Android app architecture and graphics

Activities. An Android app is comprised of multiple *activities*, which serve as an app's entry point to communicate with users and are independent units that can be called separately [6]. Each activity is implemented as a separate process by default and defines a set of callback functions invoked on specific events (e.g., `onCreate()` is called when the activity is created). When the user launches an app, one activity, denoted the *main activity*, is the first to appear [10].

Intents. Intents enable an activity (the *caller*) to invoke another activity (the *callee*). Intents can be *explicit* or *implicit*. With explicit intents, the caller specifies both the app (package) and the specific activity (callee) it wishes to invoke. With implicit intents, the caller specifies an action (e.g., sending an SMS message or opening a website), can optionally specify additional arguments to that action (e.g., the message recipient or website to be opened) and the app that should respond to the action (e.g., Google Messages or Chrome), but does not specify the activity (callee) that receives the intent. Android determines which callee receives the implicit intent based on the specified action in the intent [9]. In the case where multiple activities can respond to an implicit intent (action), the user is prompted to select which activity to invoke. Listing 1 shows sample code to send an implicit intent that opens a URL in Chrome.

Which activities can be the callee of an intent is specified by the app's *manifest file*. This file is an Extensible Markup Language (XML) file that lists all activities in the app (along with permissions and other app information). Each activity in the manifest has an `exported` attribute which determines what apps can invoke that activity. If the activity's `exported` attribute is `true`, any activity in the system can send an intent to that activity. If the activity's `exported` attribute is `false` or is undefined, only activities in the same app can send an intent to the activity [7]. Further, for activities that can receive implicit intents, the manifest file specifies one or more *intent filters*, which describe the general actions the activity can receive. Figure 1 shows an example of how a manifest file determines the allowed interactions between activities.

Tasks. In Android, a *task* represents the context through which a user interacts with a set of activities [17]. Typically, each task appears as a separate entry in the Recents screen, which allows the user to switch between open tasks. A task's *back stack* represents the history of activities the user has navigated to within a task.

Activities are stored in the back stack in a last in, first out order. For example, when the user first launches the Gmail app from the app launcher, a new task is created with Gmail's Inbox activity. We call this activity the *root activity* on the task's back stack. When the user selects an email from the Inbox, Gmail opens a new activity to view that email, and Android adds this activity to the task's

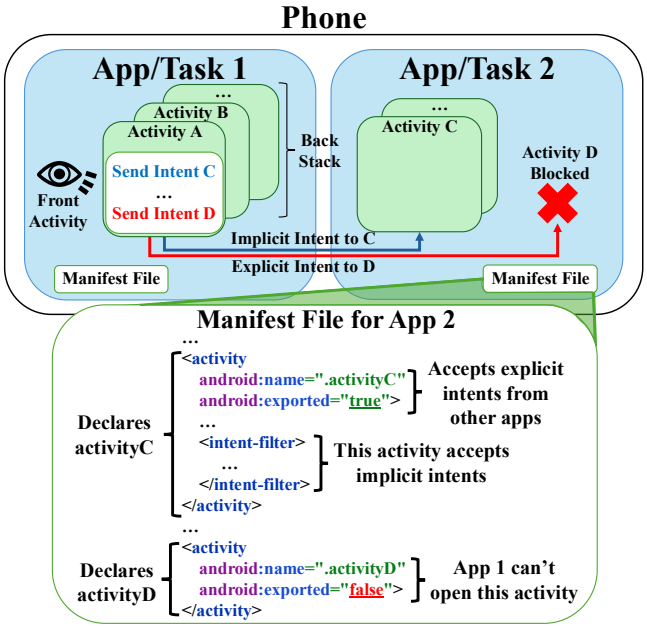


Figure 1: App 1 sends an implicit intent to `activityC` and an explicit intent to `activityD`. Since `activityC` declares an intent filter, it can receive implicit intents. However, `activityD` cannot receive intents from App 1, as the activity is marked as `exported=false`, causing the explicit intent to be rejected.

back stack. When the user presses the back button from the email viewing activity, that activity is popped off the back stack and the Inbox activity is resumed. Finally, when the user presses the back button from the Inbox (root) activity, Android resumes the task that was running before Gmail's task came to the foreground.

By default, when a callee activity is opened via an intent, Android creates a new instance of that activity in the task that invoked the intent. This occurs even when the callee belongs to a different app than the caller. However, if the callee activity has the `android:launchMode="singleTask"` attribute in the manifest file, Android does not open this activity in the same task as the caller. Instead, Android either starts a new task for the callee or, if the callee already has a task running in the background, brings that existing task to the foreground to handle the intent. In the latter case, the whole task of the callee is brought to the foreground of the caller—including the activities in the back stack of the callee.

Starting activities from the background. To reduce unwanted user interruptions, Android imposes restrictions on *when* a background activity is allowed to start another activity. Google outlines several conditions under which a caller activity is permitted to launch other activities [14]. These include, for example, when the caller has a visible window in the foreground, when the caller is in the back stack of the foreground task, or if the app has an activity that started very recently (within 10 s), regardless of whether this caller is in the back stack of the foreground task. Tuncay et al. show that this third condition allows a caller activity to retain the ability to start activities from the background *indefinitely*, by periodically starting an invisible activity every 10 s and immediately moving this activity to the background (with the `moveTaskToBack` API) [50].

Windows and views. Each activity owns a *window*, with a specified width and height, that governs where the app's pixels are drawn. A window can be transparent and as small as one pixel. A *view* is Android's basic UI building block which occupies a rectangular sub-area inside a window. For example, a view might correspond to a button, text box, or image. The activity's window houses all the activity's views which are organized hierarchically.

SurfaceFlinger. When multiple activities are visible simultaneously on the screen, Android combines their windows together in a process called *composition*. This process is handled by an Android service called *SurfaceFlinger*. By default, SurfaceFlinger treats all windows being composed as having the same upper-left anchor point on the screen. After composition, SurfaceFlinger is responsible for sending the final composited screen to the display [15]. We note that *all* operations related to the composition of multiple windows (e.g., window blurs), which we refer to as *cross-window operations*, are handled by SurfaceFlinger using GPU shaders.

SurfaceControl. Each activity has a single root `SurfaceControl` that represents its window in the Android rendering pipeline. This `SurfaceControl` defines various properties of the activity's window—such as its size, blur radius, and upper-left anchor point—that are used by SurfaceFlinger when composing the screen. Although not directly accessible to developers, `SurfaceControl` can be partially configured through the window object returned by `Activity.getWindow()`. For example, calling `getWindow().setBackgroundBlurRadius(blurRadius)` applies a blur effect to the window by updating the underlying `SurfaceControl`.

VSync signals. In Android, VSync signals are used to synchronize three key phases of the rendering pipeline: when activities perform rendering, when SurfaceFlinger wakes up to composite multiple windows, and when the display refreshes [19]. These signals originate from the display subsystem and are typically generated at a fixed rate matching the screen refresh rate—for example, every 16.6 milliseconds on a 60 Hz display. In response to each VSync signal, Android schedules rendering work for activities that are both visible and need to be redrawn (e.g., due to UI updates). Additionally, Android allows activities to register a callback that is invoked after each VSync signal, giving activities an opportunity to execute custom logic at the start of each new frame.

2.2 Android hidden API

Android supports many public APIs that developers can use to build apps. However, it also features several methods that are only used internally by Android and are meant to be inaccessible to the developer at compile time. These methods, referred to as the *hidden API*, are annotated with the `@hide` Javadoc attribute in the Android source code. At the time of writing, there exist several bypass mechanisms that allow developers to use Android's hidden API—for example by using Java's Unsafe APIs [56].

2.3 Browser pixel stealing attacks

Embedding content from other pages is a core feature of the web, most directly supported by iframes which allow rendering a (potentially mutually distrusting) page within another. Because of this, the embedding website must be prevented from directly accessing the contents of the embedded website (and vice versa). This is achieved via the *Same-Origin Policy (SOP)*, disallowing site/origin A from accessing data from site/origin B. It is then somewhat surprising that browsers have long included structured vector graphics (SVG) filters—applied via CSS—for custom visual effects on arbitrary HTML elements, *including* iframes embedding other websites.

In 2013, Paul Stone [48] demonstrated that using specific combinations of SVG filters, an attacker can mount a side-channel attack to recover the pixels of such an iframe. Stone observed that some of these SVG transformations had pixel color-dependent execution time—creating a timing side channel that can be used to leak pixel colors when applied to an iframe the framing website cannot directly interact with. The attack proceeds in three steps:

(1) First, the attacking website uses iframes to embed a victim website. This enables the attacker to submit victim pixels to the browser's rendering pipeline.
(2) Second, the attacker applies a sequence of styling tricks and SVG filters on the victim iframe. This enables them to (semi-arbitrarily) *compute* on individual victim pixels.
(3) Third, the attacker measures side channels created by Step 2's computation—e.g., by measuring the time to render browser frames using the `requestAnimationFrame` API (invoked when page rendering completes).

There is significant flexibility in how Step 2 is implemented, as there are many different SVG filters ranging from convolution, to blur, to displacement maps, and they may be layered or composed.

A challenge in Stone's attack is that the attacker needs to simultaneously compute over an individual victim pixel at a time and induce a large enough timing difference to be measurable by Step 3. Stone solves this by using SVG filters and other features as follows:

(1) A color transformation filter binarizes the image to black and white pixels.
(2) Another layered element masks out all but one target pixel.
(3) The attacker uses styling options to mirror the target black-or-white pixel into an enlarged $N \times N$ area.
(4) The region is multiplied against a noisy image to obtain a uniform black or noisy result.
(5) Finally, an SVG filter with data-dependent timing (e.g., `feMorphology`) is applied.

By setting N large (e.g., $N = 100$), the timing difference created by the final filter is made practical to measure.

Stone's work inspired many follow-on pixel stealing attacks [21, 28, 34, 42, 43, 49, 51, 53]. All these subsequent attacks utilize Stone's techniques and differ primarily in what causes the timing side channel generated in Step 2 and measured in Step 3. For the remainder of the paper, we refer to all pixel stealing attacks that utilize Stone's browser-based framework as *Stone-style pixel stealing attacks*.

Fortunately, Stone-style pixel stealing attacks are largely mitigated by major browsers and websites today. Specifically, the `X-Frame-Options` header and the `frame-ancestors` content-security-policy (CSP) directive allow sensitive websites to prevent being framed by other websites, blocking Step 1 in Stone's attack. Additionally, by default, modern web browsers restrict cookies from being sent with cross-origin requests. For example, Firefox uses Total Cookie Protection [33, 37], which partitions cookie storage by the top-level domain being visited, Safari blocks third-party cookie access by default [54], and Chrome enforces the `SameSite=Lax` cookie policy, which prevents cookies from being sent for cross-origin frame requests unless `SameSite` was explicitly set to `None` [2]. Without cookies, an embedded iframe is unlikely to display sensitive information. Taken together, the above mean that Stone-style pixel stealing attacks impact very few websites today.

2.4 GPU.zip side channel

GPU.zip is a side channel exploiting graphical data compression in modern GPUs [51]. Graphical data compression has been deployed widely in mobile devices as increasing screen sizes drive up the per-frame DRAM traffic. Such compression reduces DRAM traffic by identifying redundancy in each frame and storing/transferring those frames between the CPU and GPU in a compressed format. To compress, the entire frame is broken into smaller, individually compressed, blocks along with additional metadata about the compression status of each block. As the compression is software transparent, it must be lossless. These lossless schemes are known to result in a data-dependent compression ratio and hence a data-dependent amount of data to transfer. Since the memory bandwidth is a limited resource, data-dependent DRAM traffic then translates to a data-dependent rendering time or can otherwise be monitored by a co-located attacker observing contention.

3 Android Pixel Stealing Framework

In this section, we present our pixel stealing framework for Android. Our key observation is that Android APIs enable an attacker to instantiate a Stone-style pixel stealing framework outside of the browser. Recall from Section 2.3 that existing browser mitigations block pixel stealing on most websites today. Our framework bypasses all these browser-based mitigations. Further, our framework is able to, for the first time, leak pixel values from *non-browser apps*.

Overview. The next three sections are organized to mirror the three main steps in Stone-style attacks (Section 2.3). First, Section 3.1 (mirroring Step 1) describes how an attacker can cause the user's device to render victim app pixels. Section 3.2 (mirroring Step 2) describes activities which isolate, enlarge, and compute on a target pixel submitted for rendering. Lastly, Section 3.3 discusses side-channel measurement capabilities on Android relevant to Step 3. The above steps are shown visually in Figure 2.

Some parts of our framework, e.g., the enlargement activity (Section 3.2) and the side channel used, require device-specific considerations. These are described in Section 4.

Threat model. We use the standard phishing and tapjacking attack threat model [22, 25–27, 30, 31, 39, 45, 50, 55]. That is, to mount our attack, the attacker must convince the user to install and run an Android app containing their attack code (either in the main app or an included library). We call this app the *attacker app*. Unlike phishing and tapjacking, however, we assume the attacker's goal is to leak pixel values displayed by other locally installed apps.

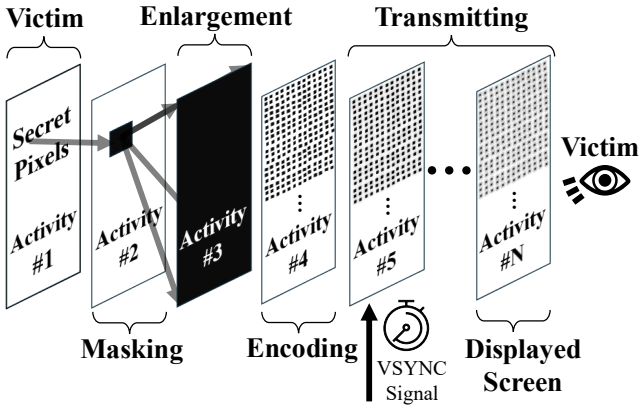


Figure 2: Overview of our pixel stealing framework. The leftmost/rightmost activity in the figure is at the bottom/top of the activity stack. The activity at the top of the stack is considered the layer closest to the user's eyes.

The attacker app does not use any Android permissions (i.e., no permissions are specified in its manifest file). After the user runs the attacker app, we make no further user-interaction assumptions.

3.1 Step 1: Submitting victim pixels to the Android rendering pipeline

In step 1 of a pixel stealing attack, the attacker must trick the user's device into sending victim app pixels to the rendering pipeline. Stone-style pixel stealing attacks achieve this step in a browser context by loading victim websites in iframes. In contrast, our framework achieves this step by sending Android intents to a victim app's activities. Recall from Section 2.1 that when a victim activity receives an intent, its associated window is opened and sent to the Android rendering pipeline. As we demonstrate below, this places all pixels in the victim activity's window at risk.

Mitigations to Stone-style attacks work by preventing sensitive websites from being embedded in iframes or preventing sensitive website content from being shown inside iframes. Our framework bypasses all these mitigations, as it does not rely on iframes and can target any content displayed in a victim activity's window. This includes sensitive website content displayed in browser apps as well as sensitive data *only* displayed in non-browser apps, e.g., Google Maps' timeline, private chat messages, ephemeral 2FA codes and more. The only requirement is that the victim activity must be openable by the attacker app via intents, as we discuss next.

Reachable victim activities. Recall from Section 2.1 that for a victim activity to be openable by other apps, it must have the `exported` attribute set to `true` in the victim app's manifest file. Fortunately, all standalone Android apps have at least one exported activity (generally the main activity) so that the top-level launcher can open them. Hence, *at least one activity in each victim app is always reachable from the attacker app via explicit intents.*[1] Further, activities that specify both the `exported=true` attribute and intent filters are also reachable from the attacker app via implicit intents. Recall from Section 2.1 that implicit intents support optional additional arguments that can *influence the behavior of the callee activity*. Since the caller is the attacker, all of this information is attacker-controlled. This enables the attacker to gain deeper access to the victim relative to explicit intents. For example, the attacker can direct a browser app to a specific URL or the SMS app to a specific conversation.[2]

Determining if a victim app is installed. Recall that no permissions are required for the attacker app to send intents to a victim activity. However, the attacker app does not know a priori what apps are installed on the device. This is an issue because if the attacker tries to send an intent to an activity of an app that is not installed, the user gets a warning saying that the attacker app "stopped working". We bypass this warning by launching intents in a try-catch block. This way, if the victim app is not installed, the resulting exception is handled gracefully without crashing the attacker app.

Of independent interest, we also observe that when the attacker sends an intent to an *installed* victim app, the attacker can prevent the victim app from appearing on screen as follows. First, the attacker sends an intent to the victim activity with `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`, which keeps the victim app hidden from the Recents screen. Second, it sends an intent with `CATEGORY_HOME`, which prevents back button presses from reaching the victim app (this is only needed if the victim activity opens in a separate task). Lastly, the attacker sends itself an intent with `FLAG_ACTIVITY_CLEAR_TOP`, which brings the attacker app back to the foreground. When the attacker sends these intents back-to-back, we observe that neither the victim app nor the Home screen become visible, and the attacker app remains on screen the entire time.[3]

The combination of the above two techniques—the try-catch block and the stealthy handling of installed app launches—allows an attacker to determine if an arbitrary app is installed on the device without triggering any warning (if the app is not installed) and without visibly opening the app being queried (if the app is installed). This finding is of interest beyond the scope of pixel stealing. For example, it could be used by attackers for user profiling. We stress that querying the list of all installed apps on a user's device is explicitly disallowed for developers since Android 11 [13].[4]

3.2 Step 2: Computing on victim pixels

In step 2 of a pixel stealing attack, the attacker must be able to perform computations on individual victim pixels that were submitted to the rendering pipeline. Stone-style pixel stealing attacks achieve this step by applying a stack of SVG filters that isolate, binarize, enlarge, and transmit individual victim pixels. In contrast, our framework achieves this step by opening *a stack of semi-transparent activities* that SurfaceFlinger composites with the victim activity. As we show, this composition process enables an attacker to achieve

[1] In Section 5.2, we survey 99,592 Android apps and show that these apps typically have additional exported activities beyond their main activity.

[2] Should a victim app define identical intent filters for multiple activities, the user would be prompted to select which activity should receive the implicit intent (Section 2.1). This would require undesired user interaction. In practice, however, we have not encountered any app where an intent filter matches multiple activities in that app.

[3] We hypothesize that the victim activity or Home screen may briefly become visible in the rare cases they start rendering *before* the third intent brings the attacker app back to the foreground, but we never observed this in our experiments.

[4] The exception is for apps that specify the restricted, "high-risk" `QUERY_ALL_PACKAGES` permission in their manifest file (subject to Google's approval) [18].

similar capabilities to Stone's SVG filter stack. The process is illustrated in Figure 2. For now, we assume the attacker's goal is to determine whether individual victim pixels are white or non-white. We discuss leaking other colors later in this section.

Activity stack. Recall from Section 2.1 that when a caller activity sends an intent to a callee activity, Android moves the callee activity to the foreground (along with its task's back stack if `android:launchMode="singleTask"`) and moves the caller activity to the background. However, despite no longer being in the foreground, the caller activity is still allowed to send intents that start additional activities from the background. For example, the caller activity can send another intent to launch a second callee activity. In this case, the second callee moves to the foreground, while the first callee is moved to the background. Further, SurfaceFlinger treats the window of the second callee as being overlaid in front of the window of the first callee. In our framework, the attacker app leverages this behavior to layer a *stack of semi-transparent activities* in front of a newly launched victim activity. In the following, we describe how the attacker uses this stack and SurfaceFlinger's APIs to isolate, enlarge, and transmit individual pixels from the victim activity.

Isolating a victim pixel. The attacker's first goal is to isolate individual victim pixels that were submitted to the Android rendering pipeline. Our framework accomplishes this goal as follows. First, the attacker opens an attacker-controlled activity—the *masking activity*—on top of the victim activity. Since the masking activity is opened after the victim activity, Android places the masking activity in the foreground and moves the victim activity to the background. Next, the attacker app makes all its existing activities transparent, except for the masking activity. This way, if the victim activity and the masking activity are in different tasks, all activities positioned between them become invisible.[5] Finally, the attacker sets the masking activity's window to be all opaque white pixels except for the pixel at the attacker-chosen location which is set to be transparent. When SurfaceFlinger composites the masking activity, any potential transparent activities between the masking activity and the victim activity, and the victim activity, the result is a window of all white pixels except for the pixel at the attacker-chosen location, whose value is set to the victim pixel's color.

Enlarging a victim pixel. The attacker's second goal is to enlarge the victim pixel isolated by the masking activity. Stone-style attacks achieve this using SVG filters applied via CSS transforms (e.g., "scale"). However, SurfaceFlinger's APIs do not support such transforms and—on the devices we tested—implement only one cross-window operation: *window blur* [20] (or, simply, *blur*).

A naive enlarging approach might be to apply the blur effect—possibly multiple times—directly to the composited masking window. However, since all the pixels surrounding the target pixel in this window are white, repeated blurring causes the target pixel's color to quickly fade to white, regardless of its original value.

To overcome this challenge, our framework leverages a subtle quirk in how SurfaceFlinger implements blur. Specifically, we observe that when blur is applied to a window with a blur radius of at least 10 px, and the blur output is restricted to a sub-region of the window, SurfaceFlinger effectively produces a stretch-like effect, where the pixels along the perimeter of the blurred region are extended outward, filling the rest of the window.

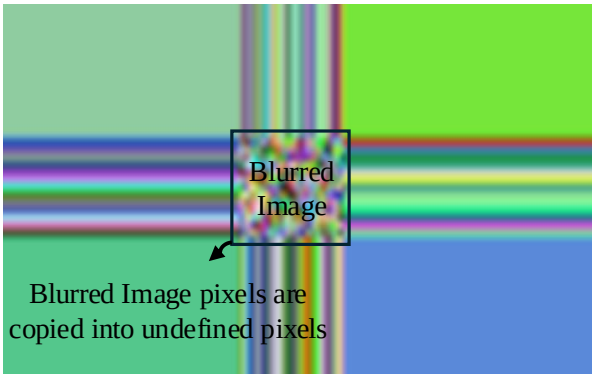


Figure 3: An subtle quirk in the way SurfaceFlinger implements blur enables a stretch-like effect to the pixels along the perimeter of the blurred pixel window. Our framework leverages this observation for pixel enlargement.

Why does this stretch-like effect occur? SurfaceFlinger's blur effect is implemented using bilinear interpolation. When blur is applied to an $N \times N$ window but the blur output is restricted to an $M \times M$ sub-region of that window,[6] SurfaceFlinger uses interpolation to estimate pixel values outside of the blurred sub-region, before storing the result onto blur's $N \times N$ output buffer. For pixels that are in the $N \times N$ window but outside the $M \times M$ blurred sub-region, interpolation assigns the value of the nearest edge pixel from the blurred region. This behavior is illustrated in Figure 3.

Our framework exploits this observation to enlarge the victim pixel isolated by the masking activity. To do so, the attacker opens a new attacker-controlled activity—the *enlargement activity*—on top of the masking activity. This new activity is fully transparent; hence, when SurfaceFlinger composites it with the masking activity, the result looks like the masking activity. Then, the attacker applies the blur effect with a blur radius of 10 px to the enlargement activity, while restricting the blur output to a 1×1 sub-region that overlaps with the victim pixel. As a result, that 1×1 region is filled with a blurred (and slightly dimmed) version of the victim pixel and—due to bilinear interpolation—all other pixels in the enlargement window inherit the color of this single blurred pixel.

Transmitting a victim pixel. The attacker's third goal is to perform computations on an isolated and enlarged victim pixel to *transmit* its value to Section 3.3's side-channel receiver. Stone-style attacks achieve this by repeatedly performing data-dependent SVG filter operations such as `feMorphology` on the enlarged victim pixel. In contrast, our framework achieves this by repeatedly applying data-dependent blur effects on the enlarged victim pixel.

Recall that the attacker's goal is to determine whether the victim pixel is white or non-white. Our framework enables this as follows. First, the attacker opens a new semi-transparent activity—the *encoding activity*—in front of the enlargement activity. The purpose

[5] Recall from Section 2.1 that when a callee that belongs to a different task than the foreground task is opened, the whole task of the callee is brought to the foreground.

[6] How to restrict the blur output to an $M \times M$ sub-region of the window is device specific and explained in Section 4.

of the encoding activity is to encode a single bit of information—whether the victim pixel is white or not—into different patterns on the screen. Second, the attacker opens a stack of transparent *transmitting activities* that apply blur effects over the encoding activity. The purpose of the transmitting activities is to transmit the encoded bit over a side channel, assuming the rendering pipeline includes pattern-dependent optimizations that leak information. The exact configuration of the encoding and the transmitting activities depends on the specific side channel being exploited. We defer details on these activities to Section 4.

Leaking non-white victim pixel colors. So far, we have assumed the attacker app's goal is to determine whether a victim pixel is white or non-white. This is sufficient in most pixel stealing attacks that are trying to recover text. More generally, however, our framework can determine whether a victim pixel is of an arbitrary color c or non-c. To do so, the attacker app can simply change the opaque pixels of the masking activity and of the encoding activity to c.

3.3 Step 3: Measuring the side effects of computations on victim pixels

In step 3 of a pixel stealing attack, the attacker must be able to measure the side effects of step 2's secret-dependent computations. Suppose that during the rendering of step 2's transmitting activities, some information about the victim pixel color is leaked via a side channel. How can an attacker app learn this information? In this section, we discuss two attacker capabilities that our framework supports to facilitate this step. These capabilities are agnostic to the side channel being exploited. As in Stone-style attacks, the details of how these capabilities are used in practice depends on the specific side channel being exploited—which we discuss in Section 4.

Executing native code. The first attacker capability supported by our framework is the ability to run native code. Specifically, the attacker app's activities can include Java, Kotlin, C/C++, and even assembly code that performs side-channel measurements asynchronously with the rendering of the transmitting activities. The ability to run native code enables the attacker to exert direct control over memory management and access fine-grained timers—both of which are useful capabilities in mounting side-channel attacks and are more powerful than what is supported by browser-based Stone-style pixel stealing attacks, which are limited to running code in managed language runtimes (e.g., JavaScript and WebAssembly).

Measuring rendering time. The second attacker capability supported by our framework is the ability to measure the *rendering time per frame* (referred to as *rendering time*). This is enabled by Android's VSync signals. Recall from Section 2.1 that these signals are sent at regular intervals to activities that are both visible and need to be redrawn (e.g., due to UI updates). Android also allows an activity to register a callback that is triggered on each VSync signal. Importantly, the timing of this callback—conceptually similar to `requestAnimationFrame` in JavaScript—can reflect how long the system took to render the previous frame, since the callback is triggered only once the system is ready to render the next frame.

In our framework, the attacker app registers a VSync callback on the rearmost transmitting activity (the one positioned behind all other transmitting activities in the activity stack). Each time the callback is invoked, it calls `View.invalidate()`, marking the activity's window as needing to be redrawn and ensuring it continues to receive future VSync signals. Invalidating this window also causes Android to invalidate the windows of all the transparent transmitting activities layered in front of it. This forces SurfaceFlinger to recomposite these windows. As a result, the next VSync callback is not invoked until SurfaceFlinger finishes compositing and rendering these windows, or until the next display refresh interval—whichever occurs later. This enables the attacker to observe delays caused by the rendering of the transmitting activities.

4 Framework Instantiations

In this section, we describe two possible instantiations of Section 3's pixel stealing framework. Specifically, Section 4.1 describes an instantiation of our framework on four recent Google Pixel phones and Section 4.2 describes an instantiation of our framework on a Samsung Galaxy S25 phone. Due to the different hardware and graphics stacks used by these devices, the two instantiations differ in the way Step 2 of the framework is implemented. However, Steps 1 and 3 are implemented the same way. In particular, for Step 3, both instantiations use the rendering time measurement capability and target victim pixel color-dependent rendering time differences due to pattern-dependent optimizations (e.g., GPU.zip [51]). In Section 4.3, we benchmark both framework instantiations using the 48×48 pixel checkerboard from Andrysco et al. [21].

Experimental setup. We conduct this section's experiments on five devices: Google Pixel 6, Google Pixel 7, Google Pixel 8, Google Pixel 9, and Samsung Galaxy S25. All phones run Android 15 and are at API level 35, the latest at the time of writing. All phones are connected to their chargers during the experiments.[7] Each experiment consists of a single app that uses the framework from Section 3 to distinguish between white and non-white pixels in a victim activity. This section's victim activities consist of either all white or all black pixels. We additionally instrument the app to prevent the screen from locking by setting the `FLAG_KEEP_SCREEN_ON` flag, which does not require any permissions.

4.1 Instantiation on Google Pixel phones

We now describe the device-specific enlargement activity and transmitting activities we use when instantiating our Android pixel stealing framework on Google Pixel 6, 7, 8, and 9 phones. All other parts of the framework remain as described in Section 3.

Enlarging a victim pixel. Recall from Section 3.2 that our framework's enlargement activity requires the ability to restrict the blur output to a 1×1 sub-region that overlaps with the victim pixel. To achieve this, our Google Pixel framework instantiation uses `SurfaceControl`'s crop API prior to applying the blur effect. Applying crop before blur requires using Android's hidden API (cf. Section 2.2). We also observe that the crop API's y-coordinate is offset relative to the y-coordinate of the activity's window. We describe how we account for this offset in Appendix A.1. We confirm that specifying the 1×1 target pixel region as input to the crop API and later applying the blur operation to the enlargement activity's window produces the desired pixel enlargement behavior.

[7] We verified that all our findings also hold when the phones were not charging.

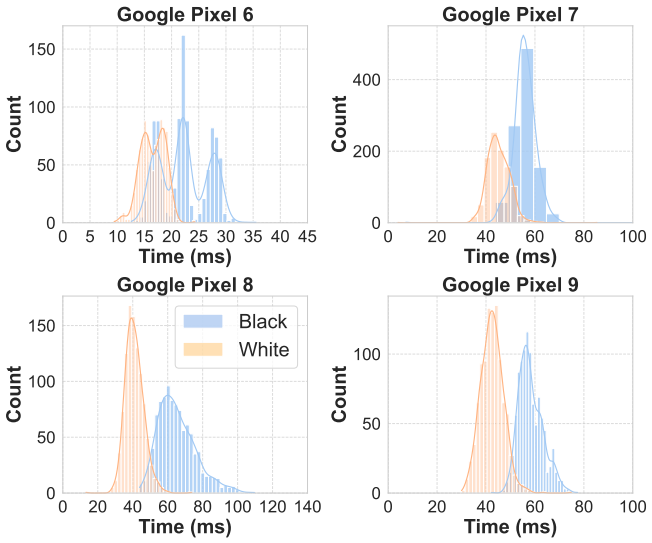


Figure 4: Rendering times for a configured activity stack in front of a black and white victim pixel on the Google phones.

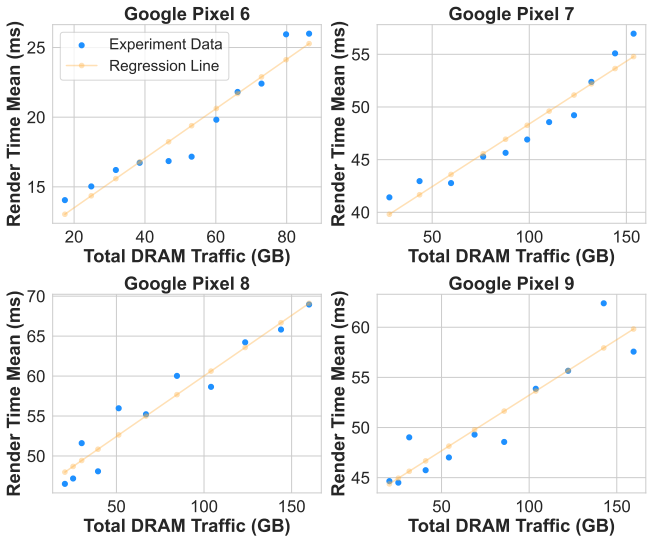


Figure 5: Rendering time vs. total DRAM traffic for the root cause analysis experiment using configured activity stacks. The data points are in ascending order with respect to the percentage of the screen covered by pattern one.

Transmitting a victim pixel. We target pattern-dependent optimizations, such as GPU.zip (cf. Section 2.4), to transmit the value of the victim pixel. Thus, the goal of our instantiation's encoding activity is to encode white pixels as a highly redundant visual pattern and non-white pixels as a highly non-redundant visual pattern. To do this, we fill the encoding activity's window with opaque white pixels but leave a number of transparent pixels randomly scattered throughout the window. This way, when SurfaceFlinger composites the encoding and enlargement activities, the result appears all white if the victim pixel is white, or white speckled with the victim pixel's color if the victim pixel is not white. We then open the stack of transparent transmitting activities—all of which apply blur effects with a 1 px blur radius on the encoded pixel. We empirically determine how many transmitting activities are needed on each device to generate an observable rendering time difference.

For each experiment, we collect 1,000 measurements of the rendering time using Section 3.3's methodology. Figure 4 shows the rendering time distributions when using 25, 55, 60, and 60 transmitting activities on our Google Pixel 6, 7, 8, and 9, respectively. We also report the average and standard deviation of rendering times in Table 1. As expected, rendering is faster when the victim pixel is white and slower when it is black. This rendering time difference is consistent with prior work's observations on the side effects of GPU graphical data compression [51], which compresses redundant patterns more efficiently than non-redundant ones.

Root cause analysis. We confirm the existence of GPU graphical data compression on our Google Pixel phones by reverse engineering the compression algorithm on their Mali GPU. We present the reverse engineered Mali compression algorithm in Appendix B.

We additionally present evidence attributing the root cause of the rendering time side channel we exploit on Google Pixel phones to graphical data compression. First, we design two encoding activity patterns: pattern one places 1×1 transparent dots 1 px apart, and pattern two places them 16 px apart. In both cases, a white victim pixel results in a white screen, while a black victim pixel results in a speckled screen. Based on our reverse engineering, the encoding output of a white victim pixel is compressible under both patterns; however, given a black victim pixel, the encoding output is not compressible with pattern one, but compressible with pattern two.

We then rerun our rendering time experiment from above using a black victim pixel and a combination of these two encoding patterns. Specifically, we vary the proportion of the encoding activity covered by pattern one from 0% to 100% (in 10% increments) and fill the remaining area with pattern two. For each configuration, we collect 200 measurements of the rendering time along with the total DRAM traffic (via Arm's Performance Studio). Because pattern one produces incompressible output while pattern two produces compressible output, we expect DRAM traffic to increase linearly as the share of pattern one increases. Further, if the rendering time side channel is indeed due to compression, we expect the rendering time to also increase linearly. Figure 5 shows the results. The R-squared value for how well DRAM traffic predicts average rendering time are 0.92, 0.92, 0.94, and 0.86 for the Google Pixel 6, 7, 8, and 9, respectively, matching our expectation. We use pattern one as the encoding pattern for the remainder of this paper.

To rule out other potential root causes such as effects due to power and thermal throttling [28, 49, 52, 53], we monitor additional performance counters including GPU frequency, CPU frequency, GPU power, and thermal throttling status. We monitor GPU frequency by reading the `cur_freq` file in the Mali folder located at `/sys/devices/platform/`. We monitor CPU frequency and GPU power via Android Studio's Profiler, and thermal throttling status via Arm's Performance Studio. In all experiments, we observe no statistically significant difference in the GPU frequency, CPU frequency, GPU power, and thermal throttling status.

Taken together, the above observations provide strong evidence that the data-dependent rendering time side channel exploited by our pixel stealing framework instantiation on Google Pixel phones is due to the graphical data compression optimization of Mali GPUs.

4.2 Instantiation on a Samsung Galaxy S25

As with the Pixel description, we now describe the device-specific aspects of our framework instantiation on our Samsung Galaxy S25. All other parts of the framework remain as described in Section 3.

Isolating a victim pixel. On our Samsung Galaxy S25, the blur applied during pixel enlargement causes enlarged non-white pixels to appear excessively white, making it difficult to distinguish between a white and non-white pixel. To mitigate this, in the masking activity of our instantiation, we use a 2×2 transparent region instead of a 1×1, thereby increasing the number of pixels being enlarged.[8]

Enlarging a victim pixel. The enlargement activity configuration used on our Google Pixel phones does not produce the desired pixel enlargement behavior on our Samsung Galaxy S25. Thus, for this instantiation, we adopt the following, different configuration. To restrict the blur output to a 2×2 sub-region that overlaps with the isolated victim pixels, we use a special Android view called `SurfaceView`. Unlike regular views, a `SurfaceView` has its own `SurfaceControl`, which is a child of the activity's root `SurfaceControl`. Further, the attacker can access this child `SurfaceControl` and configure its rendering properties. In particular, the attacker sets the `SurfaceView` scale to 0.001 along both the x and y dimensions and positions it such that the downscaled surface overlaps precisely with the victim pixels.[9] After configuring the position and scale, we apply a blur effect on the `SurfaceView`. On our Samsung Galaxy S25, the blur effect is not a public API by default (unlike on Pixel phones) and can only be set using the Hidden API. We confirm that these steps produce the desired pixel enlargement behavior.

Transmitting a victim pixel. As in Section 4.1, we target pattern-dependent optimizations to transmit the value of the victim pixel. However, we experimentally find that the transmitting activity configuration used on our Google Pixel phones does not produce pixel color-dependent rendering time differences on the Samsung Galaxy S25, even when using a large number of transmitting activities (e.g., 70). Thus, for this instantiation, we adopt the following, different configuration. In each transmitting activity, we use the hidden API to define 30 blur regions, each with a distinct blur radius ranging from 1 px to 30 px. Defining multiple regions causes multiple blur computations within a single activity. However, to ensure these computations are not skipped by SurfaceFlinger's caching mechanism, each blur region must use a unique radius.

We use this configuration and Section 3.3's measurement methodology to collect 1,000 rendering time measurements per victim pixel. Figure 6 shows the results obtained with 5 transmitting activities. As expected, rendering is consistently faster when the victim pixel is white. This demonstrates that rendering time still leaks pixel color information on the S25, despite its framework instantiation using a different configuration than the one on Google Pixel phones.

[8] Moreover, this change has minimal impact on attack accuracy (cf. Section 4.3).
[9] As in Section 4.1, we observe that the scale API's y-coordinate is offset relative to the one of the activity's window. Appendix A.2 describes how we account for this offset.

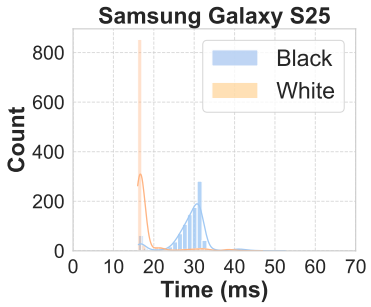


Figure 6: Rendering times for a configured activity stack in front of a black and white victim pixel on the Galaxy S25.

Table 1: Activity stack and checkerboard experiment results. Checkerboard results shown both when targeting single pixels (px/s) and when targeting 1 dp × 1 dp windows (px/s+).

	Activity Stack Test		Checkerboard Test		
	Black (ms)	White (ms)	Acc.	px/s	px/s+
Pixel 6	22.3 ± 4.4	16.6 ± 2.2	96.18%	0.23	0.91
Pixel 7	56.0 ± 5.7	45.1 ± 4.8	95.83%	0.19	0.77
Pixel 8	65.4 ± 10.8	41.0 ± 5.3	86.63%	0.19	0.77
Pixel 9	64.4 ± 4.7	52.9 ± 5.6	94.92%	0.15	0.59
Galaxy S25	29.0 ± 4.6	18.1 ± 4.6	95.14%	0.73	2.11

Root cause analysis. We leave a full root cause analysis on Samsung devices to future work, as we lack access to the necessary performance counters and cannot confidently attribute the observed timing differences to GPU graphical data compression.

4.3 Evaluation

To benchmark our pixel stealing framework, we reproduce the 48×48 checkerboard pattern used by Andrysco et al. [21]. We display the checkerboard in a victim activity and layer our framework instantiation on top. For the Pixel 6, Pixel 7, Pixel 8 and Galaxy S25, we collect 34 rendering time measurements per pixel; for the Pixel 9, we collect 64 measurements to account for increased noise. We take the median of these measurements and compare it to a timing cutoff to classify each target pixel as white (low timing) or black (high timing). This cutoff is determined during an initial calibration phase, which collects 200 measurements each for black and white pixels and sets the threshold between their median values.

To avoid noise due to throttling, we insert a 1.5-second delay between experiments on successive pixels. We report the leakage throughput (px/s) for each device in Table 1. The Samsung Galaxy S25 achieves a higher throughput than the Google Pixel phones because its implementation leaks a 2×2 pixel region at a time. We use these configurations in the remainder of the paper.

Improving speed. Generally, mobile devices have large pixel densities, and most apps use density-independent pixel (dp) units when drawing to the screen. Dp is a virtual unit that scales with screen density, where 1 dp = 1 px at 160 dots per inch. We find that 1 dp corresponds to 2.625 px on the Google Pixel phones and 3 px on the

Galaxy S25. Since dp, not pixels, generally represents the smallest graphical unit, a simple optimization to improve our attack throughput without compromising accuracy is to adjust the region leaked per measurement, denoted as resolution, to 1 dp × 1 dp. An attacker can adjust the region leaked per measurement by adjusting the transparent region in the masking activity. For the Pixels, we adjust to a 2 px × 2 px[10] resolution, and for the Galaxy S25, we adjust to a 3 px × 3 px resolution. The improved throughput after adjusting the attack resolution is shown in Table 1. We experimentally find that these resolutions have minimal impacts on our attack accuracy, and we adopt these settings for the remainder of the paper.

5 End-To-End Attacks

In this section, we evaluate the capabilities of our pixel stealing framework. Unlike Stone-style pixel stealing attacks, which affect only a limited set of websites, our framework targets a broader range of websites as well as a new security domain: non-browser Android apps. We discuss our attacks against browsers and non-browser apps in Section 5.1 and Section 5.2, respectively. For all attacks, we use the same experimental setup as Section 4.

5.1 Browsers

First, we describe how an attacker can open any target website either directly in the browser or in a Custom Tab. Then, we present a comparative survey of websites vulnerable to Stone-style pixel-stealing attacks versus those vulnerable to our framework. The results highlight that our framework poses a much broader threat. We illustrate this with three end-to-end attacks targeting Google Accounts, Gmail, and Perplexity AI, all of which are sensitive websites that were not vulnerable to Stone-style pixel stealing attacks due to iframe embedding restrictions.

Opening websites via Intents. To open a website, an attacker can send an implicit intent to any browser with the view action and set the data field to the desired target URL, as shown in Listing 1. An attacker can also create an implicit intent using the Custom Tabs API to open a target URL in a Custom Tab, a lightweight, embedded version of Chrome commonly used by Android apps to display web content in the same task without launching the full browser [12]. For all following web attacks, we perform the attack on both the Chrome browser and Chrome's Custom Tabs.

Web survey. We start by surveying and comparing the number of websites vulnerable to our framework versus Stone-style pixel stealing attacks. Recall from Section 2.3 that a victim website is vulnerable to Stone-style attacks (in Chrome) only if it is iframeble *and* it disables cross-site cookie sharing protections (`SameSite=None`). We examine these properties for the top one million websites listed in the Chrome User Experience Report (CrUX) [29, 47]. For each website, we collect whether it can be embedded in an iframe (iframable), and its `SameSite` cookie policy—categorized as `None`, `Lax`, or `Strict`. Cookies with the `None` attribute are included in all requests, including cross-site contexts, making them fully accessible across frames and tabs (in Chrome). `Lax` cookies are sent during top-level navigations using the `GET` method—such as when clicking a link—but not with requests due to iframes or images. Finally, `Strict` cookies are only sent in same-site contexts, meaning they are excluded from all cross-site requests, including top-level navigations and requests initiated via iframes or Custom Tabs [36].

[10]Floating-point resolutions are not supported.

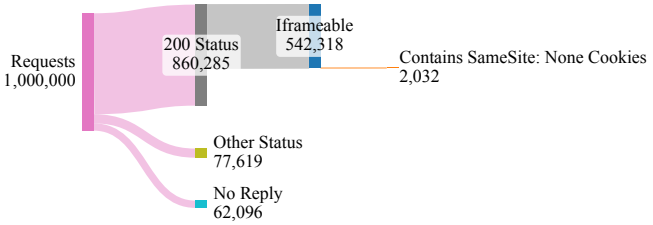


Figure 7: Results of our web survey. Only 2,032 (0.2%) are vulnerable to Stone-style pixel stealing attacks, which require both iframe embedding (iframeable) and cookies with `SameSite=None`. In contrast, all reachable websites are vulnerable to our framework.

For each website, we send a `GET` request with a current Chrome-on-macOS user-agent string. We mark a website as reachable if it returns an HTTP 200 status code, and unreachable if it does not respond within five seconds. We determine if it can be embedded in an iframe from the `X-Frame-Options` and `Content-Security-Policy` headers and extract its `SameSite` cookie policy.

We show our survey result in Figure 7. Among the top 1,000,000 websites, 860,285 are reachable. Of the reachable websites, only 0.2% are vulnerable to Stone-style pixel stealing attacks. The small set of vulnerable websites may explain why Google has deprioritized mitigating recent Stone-style attacks on Chrome [1, 3–5].

In contrast, our framework can target any website that displays private information. 100% of the reachable websites are vulnerable to our framework when being opened in the Chrome app. When opened in a Custom Tab, 99.3% of reachable websites are vulnerable; the remaining 0.7% mark cookies with `SameSite=Strict`, which are withheld on cross-site navigations initiated by an external app.

We note that several websites implement anti-crawler measures or have varying cookie-sharing states after a user logs in which affects the accuracy of our survey. For example, Wikipedia enables cross-site cookie sharing after a user is logged in, but this is not captured by our methodology. Nonetheless, this survey provides a valuable overview of the limited reach of Stone-style attacks.

Google Account. To start, we target the URL `myaccount.google.com` and recover the screen region displaying the user's full name. This region occupies 56,120–77,616 pixels on the screen (depending on the device), and our unoptimized attack requires 12–28 hours to recover all these pixels across all phones. An attacker can also use URL fragment identifiers to scroll the page to specific sections of the Google Account page, including those displaying the user's physical addresses, email addresses, account settings, account sessions, connected third party apps and services, birthday, gender, and more. For example, navigating to `myaccount.google.com/personal-info#:~:text=Addresses` scrolls to the Addresses section and displays the user's home address in a region occupying 15,600–21,546 pixels (depending on the device). Our unoptimized attack requires 3–8 hours to recover all these pixels across all phones.

Gmail. Next, we target the URL `mail.google.com`. Specifically, we recover the screen region displaying the sender, subject, and first line of text of the top email in the user's inbox. This region occupies 56,800–82,425 pixels (depending on the device), and our unoptimized attack requires 10–25 hours to recover all these pixels across all phones. An attacker can also use URL fragment identifiers to search through the user's inbox or navigate to a specific Gmail folder. For example, the `#search/social+security+number` fragment identifier finds all emails containing "social security number". Advanced search queries additionally support filters such as sender, recipient, size, and date—allowing the attacker to target and leak specific emails. Moreover, the search results page also displays a count of matching emails, which can be used to fingerprint users [32].

Perplexity AI. As a final example of a website that is vulnerable to our framework but not to Stone-style attacks, we target the URL `www.perplexity.ai/library`, which displays a user's recent chat history (topic, 2-line summary, and date of each conversation) in a screen region occupying 56,392–72,198 pixels per conversation (depending on the device). Our unoptimized attack recovers all these pixels in 10–25 hours across all phones.

5.2 Non-browser Android apps

We now evaluate the threat our framework poses to non-browser apps. As with websites, we conduct a survey to quantify the number of activities that our framework can target and then provide four end-to-end attacks targeting widely used apps that handle sensitive user data: Google Maps, Google Messages, Venmo, and Signal. We highlight that non-browser apps represent a distinct security domain, as some secrets—such as those extracted from Google Maps, Signal, and Google Messages—do not have web access. As a result, these secrets are only accessible through our framework and are fundamentally out of reach for Stone-style attacks.

App survey. We start by surveying what activities, and associated apps, are vulnerable to our framework. Recall from Section 3.1 that an activity is considered vulnerable if an attacker can complete the first step of a pixel stealing attack: sending victim pixels into the Android rendering pipeline. This can be achieved by sending an intent to a target activity belonging to a victim app.

We analyze the `manifest.xml` files from 99,592 apps on the Google Play Store using the manifest dataset from Beer et al. in TapTrap [24]. For our analysis, we exclude 2,809 apps targeting SDKs earlier than Android 13, which are not available our devices [11]. From the remaining 96,783 apps, we identify 387,958 activities that can be opened by our attacker app via explicit intents; of these, 238,036 can also be targeted by implicit intents. A median of two activities per app are exported. 109 apps do not have any exported activities; however, further analysis determined these were extensions for other apps. Overall, all the available, standalone apps we analyzed are vulnerable to our framework. Even worse, many apps have several vulnerable (`exported=true`) activities.

Google Maps. For our first end-to-end attack, we demonstrate leaking Google Maps' Timeline, which stores a user's complete *location history*, including timestamps and addresses of all visited places. This location history would include sensitive information such as home address, daily routines, and more. Since 2024, web access to Timeline has been removed due to privacy concerns [46], leaving the Google Maps app as the only access point.

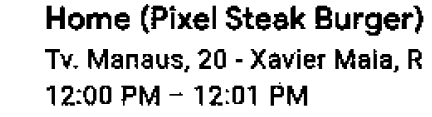


Figure 8: Recovered Google Maps Timeline entry on a Pixel 7.

We find that an attacker can open Timeline to *any date* via implicit intents and leak a user's entire location history using our framework. To do so, the attacker issues an intent to the Google Maps package (`com.google.android.apps.maps`) with the data field set to `www.google.com/maps/timeline`. For a specific date, the attacker can append a query parameter, e.g., `www.google.com/maps/timeline?pb=!1m2!1m1!1s<yyyy-mm-dd>`. Each Timeline entry is displayed in a region of 54,264–60,060 pixels (depending on the device), which our unoptimized attack takes 20–27 hours to recover across all devices. Figure 8 shows our recovered entry.

Venmo. Our second attack targets financial data displayed in the Venmo app. We observe that an attacker can use implicit intents to open any activities in the Venmo app, including those displaying account balance, transaction history, linked bank accounts, statements, and tax documents. For example, an intent with the data field set to `venmo.com/settings/profile` displays the user's Venmo username. Using the leaked username, an attacker can craft another implicit intent with the URL `venmo.com/u/<username>` to access the user's profile, which displays recent transactions and the account balance. Our unoptimized attack leaks the account balance, which is displayed in a screen region of 7,473–11,352 pixels (depending on the device), in 3–5 hours across all phones.

Google Messages. Next, we target SMS conversations displayed in the Google Messages app. Sending an explicit intent to Google Messages' main activity displays recent conversations, and sending an implicit intent can open a *specific SMS conversation* by setting the intent's data field to `smsto:<PHONENUMBER>`. For example, since 2FA codes are generally sent from a limited set of known short phone numbers, an attacker may target these numbers directly. In our evaluation, we target a received SMS message that reads "Super secret pixel stealing message" and is displayed in a screen region of 35,500–44,574 pixels (depending on the device). Our unoptimized attack recovers all these pixels in 11–20 hours across all phones.

Unlike our prior attacks (which test for white vs non-white pixels), our Google Messages attack aims to distinguish gray vs non-gray pixels (for received messages) or blue vs non-blue pixels (for sent messages). We achieve this by changing the opaque pixels of the masking and encoding activities to gray or blue, as discussed in Section 3.2. In practice, the attacker may not initially know whether a particular screen region contains a sent or a received message. Because sent messages appear on the right and received messages on the left, this can be resolved by probing (with a blue-vs-non-blue test) a few pixel locations on the right side of the conversation. If blue is detected, the region is treated as sent; otherwise it is treated as received, and the gray-vs-non-gray test is used for full recovery.

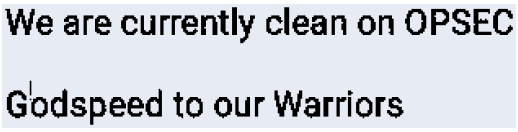


Figure 9: Recovered private Signal text message on a Pixel 7.

Signal. For our final end-to-end attack, we target the private messaging app Signal. An attacker can use implicit intents to open *any* private conversation by setting the implicit intent's data field to `sgnl://signal.me/#p/<PHONENUMBER>`. Similarly to the Google Messages attack, we aim to distinguish gray vs non-gray pixels (for received messages) or blue vs non-blue pixels (for sent messages). In our evaluation, we target a received Signal message displayed in a screen region occupying 95,760–100,320 pixels (depending on the device). Our unoptimized attack recovers all these pixels in 25–42 hours across all phones, even with Signal's Screen Security feature enabled. Figure 9 shows the recovered message.

6 Case Study: Ephemeral 2FA Code Recovery

The attacks described in Section 5 take hours to steal sensitive screen regions—placing certain categories of *ephemeral* secrets out of reach for the attacker app. Consider for example 2FA codes. By default, these 6-digit codes are refreshed every 30 seconds [38]. This imposes a strict time limit on the attack: if the attacker cannot leak the 6 digits within 30 seconds, they disappear from the screen.

In this section, we demonstrate an *optimized* end-to-end attack that reliably leaks 6-digit 2FA codes from Google Authenticator in less than 30 seconds. Our attack combines the techniques from Section 5 with a few optimizations, most notably, the OCR-style technique proposed in Stone's original paper [48]. This technique recognizes that *leaking all pixels of each secret digit is unnecessary.* Instead, assuming the font is known to the attacker, each secret digit can be differentiated by leaking just a few carefully chosen pixels. Below, we describe how we implement this technique.

We stress that the results of this Section's optimized attack (seconds instead of hours) are a more realistic representation of the attack speed than the times presented in Section 5. However, applying the OCR-style technique requires carefully selecting which pixels differentiate each digit of the secret, which is currently a manual, per-app process. While in this paper we only manually select these pixels for Google Authenticator, we do think that this process could be automated with further engineering.

Implementation. Google Authenticator displays 2FA codes using the Google Sans font. We apply Stone's OCR-style algorithm to Google Sans digits to determine which pixels differentiate each digit and find that each digit can be uniquely identified using at most 4 pixels. An example coordinate that bisects the set of digits is shown in Figure 10. Unlike the fixed-width font (Courier) targeted in Stone's work, however, Google Sans is variable-width. To address this, our attack implementation dynamically shifts the target pixel positions based on the known width of each leaked digit. Additionally, Google Authenticator inserts extra spacing between the third and fourth digits of the 2FA code (splitting the 6-digit code into two 3-digit groups). We find that this spacing varies based on the value of the third digit. After recovering the third digit, our attack implementation adjusts the pixel offset accordingly.

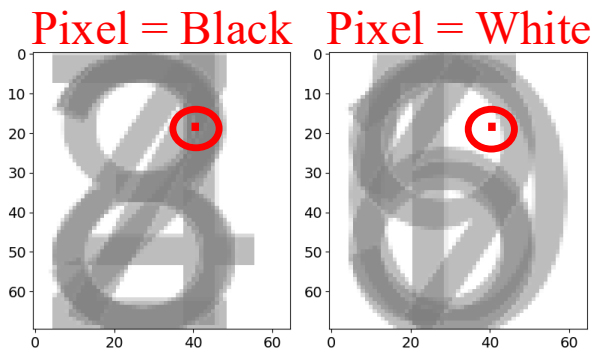


Figure 10: The left figure has digits 2, 3, 4, 7, and 8. The right figure has digits 0, 1, 5, 6, and 9. The attacker can bisect the set of possible digits in half by leaking a specific pixel's color.

To meet the strict 30-second deadline for the attack, we also reduce the number of samples per target pixel to 16 (compared to the 34 or 64 used in earlier attacks) and decrease the idle time between pixel leaks from 1.5 seconds to 70 milliseconds.[11] To ensure that the attacker has the full 30 seconds to leak the 2FA code, our implementation waits for the beginning of a new 30-second global time interval, determined using the system clock.

Evaluation. We use our end-to-end attack to leak 100 different 2FA codes from Google Authenticator on each of our Google Pixel phones. Our attack correctly recovers the full 6-digit 2FA code in 73%, 53%, 29%, and 53% of the trials on the Pixel 6, 7, 8, and 9, respectively. The average time to recover each 2FA code is 14.3, 25.8, 24.9, and 25.3 seconds for the Pixel 6, Pixel 7, Pixel 8, and Pixel 9, respectively. We are unable to leak 2FA codes within 30 seconds using our implementation on the Samsung Galaxy S25 device due to significant noise. We leave further investigation of how to tune our attack to work on this device to future work.

7 Discussion

Attack stealth. As we have described it, our attack framework generates graphical artifacts visible to the user. These artifacts can be hidden, with no significant effect on attack speed or accuracy, by layering a slightly transparent (less than 1%) hiding activity over the attack. The hiding activity can display benign content consistent with the app's claimed function. Figure 11 shows what the user sees when the attack uses a hiding activity.

Attack limitations. Our attack relies on a hidden API bypass in both framework instantiations (cf. Section 4). Hidden API bypasses are known for all recent Android versions. Early approaches using Java Reflection were blocked in Android 9; the current widely used bypass takes advantage of Java's `Unsafe` class [56]. Other bypass techniques may exist, and it is also possible that our framework could be instantiated using only the intended SurfaceFlinger API.

Our demonstrated leakage rate, 0.6 to 2.1 pixels per second, is also low (though sufficient to recover Authenticator codes). Experience

[11] On the Pixel 6, we use 24 trials.

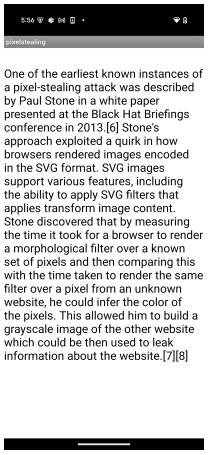


Figure 11: User's view when using a slightly transparent (less than 1%) hiding activity over the attack.

with browser pixel stealing suggests that more carefully optimized exploits that rely on rendering time as the leakage channel could approach a rate of one pixel per screen refresh; exceeding this bound would require a higher-capacity leakage channel [43].

Web history sniffing. It has long been assumed that browser pixel stealing also enables history sniffing: the attacker embeds links on their own page, then uses pixel stealing to determine whether the browser styled them as visited or unvisited. Once Chrome's history partitioning mechanism rolls out [8], links will remain unvisited on the attacker's site regardless of the user's history. With our framework, an attacker could nevertheless check whether a user has visited site A from site B (say, Google) by embedding and stealing pixels from a page B that contains a link A.

Mitigations. Recall that there are three conditions required for our pixel stealing framework. Mitigating any one of the conditions will stop our attacks. Experience with browser pixel stealing suggests that condition three should *not* be the focus of mitigation: attackers are likely to discover and apply a different side channel if ours is patched. Opening and layering activities is a core Android feature, so changes that target condition one may not be acceptable to users.

We therefore believe that our attack would be best mitigated by targeting condition two, i.e., by preventing attacker computations on victim pixels. One way to achieve this would be to allow developers to restrict transparent layering over their activities to an explicit allowlist. An analogous mechanism, the frame-ancestors CSP directive, defanged Stone-style attacks on the web by allowing sites to opt out of framing or to restrict framers to an allowlist. One other way to achieve this would be to allow developers to hide sensitive visual content when layering occurs.

8 Related Work

Pixel stealing attacks. Barth [23] was the first to observe, in 2011, that CSS filters applied to cross-origin pixel data might leak that data via a timing channel. Two years later, Stone [48] (and, independently, Kotcher et al. [35]) identified an SVG filter whose implementation in Firefox exhibited color-dependent timing variation and showed how the timing variation could be amplified and measured, proving that Barth was correct. Follow-up work instantiated Stone's framework with other violations of constant-time programming principles in the browser graphics stack [21, 28, 34, 42, 43, 49, 51, 53]. Of these, Wang et al. [51] are closest to our work in taking advantage of data-dependent graphical data compression. Our work is the first to translate Stone's ideas from the browser to mobile apps, making it possible to target all websites (not just those that agree to be framed) as well as native apps.

Custom Tabs security. Palfinger et al. [44] initiated the study of the security implications of Android Custom Tabs. They showed that an attacker app could deduce whether the user was logged onto a site loaded in a Custom Tab by measuring the time taken to load the website. Beer et al. [25] investigated the Custom Tab API, showing that it could be used to violate integrity and confidentiality guarantees of a loaded website (though they did not demonstrate pixel stealing via the API). To steal pixels from a website, our attack can load it either using a Custom Tab or directly in the browser app, bypassing the Custom Tabs API.

Mobile UI attacks. Extensive work has also been done on phishing [22, 26, 27, 30, 50] and tapjacking (aka clickjacking) [31, 39, 45, 55] attacks. These attacks take advantage of user confusion between safe and unsafe UI contexts to elicit unsafe and unintended actions. While our attacks rely on overlaying the attacker app over a target app, they exploit properties of the Android graphics stack and do not rely on user confusion or interaction (beyond installing and launching the attacker app).

9 Conclusion

Stone's story of knocking iframes and SVG filters together to steal pixels hasn't really changed with a decade of attacks. The particulars shifted as side channels came and went while restrictions from `X-Frame-Options`, `frame-ancestors`, and browser mitigations increasingly prevented the worst outcomes.

This pattern obscured the true lesson of the story: letting adversaries compute on your pixels *will* leak them.

Here, we have evolved pixel stealing beyond the constrained iframe attack model to the full range of mobile apps with predictable, and concerning, consequences. Like browsers at the beginning, the intentionally collaborative and multi-actor design of mobile app layering makes the obvious restrictions unappealing. App layering is not going away, and layered apps would be useless with a no-third-party-cookies style of restriction. A realistic response is making the new attacks as unappealing as the old ones: allow sensitive apps to opt-out and restrict the attacker's measurement capabilities so that any proof-of-concept stays just that.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We also thank Philipp Beer and Marco Squarcina for the fruitful conversations on their Tabbed Out work [25] and for sharing the manifest dataset from their TapTrap work [24] with us. This work was funded by ARL grant W911NF-25-1-0179, NSF grants 1942888, 1954521, 2120642, 2120696, 2153388, and 2154183, and gifts from Google, Intel, Mozilla, and Qualcomm.

Availability

We have open sourced all of this paper's artifacts at https://github.com/TAC-UCB/pixnapping.

References

[1] 2017. Canvas composite operations and CSS blend modes leak cross-origin data via timing attacks. Online: https://issues.chromium.org/issues/40086984. Accessed on October 9 2025.

[2] 2020. Feature: Cookies default to SameSite=Lax. Online: https://chromestatus.com/feature/5088147346030592. Accessed on April 10 2025.

[3] 2023. Security: Cross-origin pixel reading via SVG filter data-dependent GPU memory bandwidth usage. Online: https://issues.chromium.org/issues/40063650. Accessed on October 9 2025.

[4] 2023. Security: Timing Attack Using SVG Filters on Visited Links to Sniff History (Based on GPU-Zip). Online: https://issues.chromium.org/issues/40943106. Accessed on October 9 2025.

[5] 2024. Cross-origin pixel reading via data-dependent SVG filter power usage and Intel Thread Director. Online: https://issues.chromium.org/issues/381109468. Accessed on October 9 2025.

[6] 2025. Activity | API reference | Android Developers. Online: https://developer.android.com/reference/android/app/Activity. Accessed on April 10 2025.

[7] 2025. <activity> | App architecture | Android Developers. Online: https://developer.android.com/guide/topics/manifest/activity-element. Accessed on April 10 2025.

[8] 2025. Feature: Partitioning :visited links history. Online: https://chromestatus.com/feature/5101991698628608. Accessed on April 3 2025.

[9] 2025. Intents and intent filters | App architecture | Android Developers. Online: https://developer.android.com/guide/components/intents-filters. Accessed on April 10 2025.

[10] 2025. Introduction to activities | App architecture | Android Developers. Online: https://developer.android.com/guide/components/activities/intro-activities. Accessed on April 10 2025.

[11] 2025. Meet Google Play's target API level requirement | Other Play guides | Android Developers. Online: https://developer.android.com/google/play/requirements/target-sdk. Accessed on September 10 2025.

[12] 2025. Overview of Android Custom Tabs | Views | Android Developers. Online: https://developer.android.com/develop/ui/views/layout/webapps/overview-of-android-custom-tabs. Accessed on April 10 2025.

[13] 2025. Package visibility filtering on Android | App architecture | Android Developers. Online: https://developer.android.com/training/package-visibility. Accessed on September 10 2025.

[14] 2025. Restrictions on starting activities from the background | App architecture | Android Developers. Online: https://developer.android.com/guide/components/activities/background-starts. Accessed on April 10 2025.

[15] 2025. SurfaceFlinger and WindowManager | Android Open Source Project. Online: https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager. Accessed on April 10 2025.

[16] 2025. SVG and CSS filters can leak cross-origin data via iframes. Online: https://issues.chromium.org/issues/401081629. Accessed on October 9 2025.

[17] 2025. Tasks and the back stack | App architecture | Android Developers. Online: https://developer.android.com/guide/components/activities/tasks-and-back-stack. Accessed on April 10 2025.

[18] 2025. Use of the broad package (App) visibility (QUERY_ALL_PACKAGES) permission. Online: https://support.google.com/googleplay/android-developer/answer/10158779. Accessed on April 10 2025.

[19] 2025. VSync | Android Open Source Project. Online: https://source.android.com/docs/core/graphics/implement-vsync. Accessed on April 10 2025.

[20] 2025. Window blurs | Android Open Source Project. Online: https://source.android.com/docs/core/display/window-blurs. Accessed on April 10 2025.

[21] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *S&P*.

[22] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In *CCS*.

[23] Adam Barth. 2011. Timing Attacks on CSS Shaders. Online: https://web.archive.org/web/20120207083807/http://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html. Accessed on March 24 2025.

[24] Philipp Beer, Marco Squarcina, Sebastian Roth, and Martina Lindorfer. 2025. TapTrap: Animation-Driven Tapjacking on Android. In *USENIX Security*.

[25] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. 2024. Tabbed Out: Subverting the Android Custom Tab Security Model. In *S&P*.

[26] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *S&P*.

[27] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *MobiSys*.

[28] Inwhan Chun, Isabella Siu, and Riccardo Paccagnella. 2025. Scheduled Disclosure: Turning Power Into Timing Without Frequency Scaling. In *S&P*.

[29] Zakir Durumeric. 2025. Chrome (CrUX) Top Million Websites. Online: https://github.com/zakird/crux-top-lists. Accessed on April 10 2025.

[30] Adrienne Porter Felt and David Wagner. 2011. Phishing on mobile devices. In *W2SP*.

[31] 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*.

[32] Nethanel Gelernter and Amir Herzberg. 2015. Cross-site search attacks. In *CCS*.

[33] Johann Hofmann and Tim Huang. 2021. Introducing State Partitioning. Online: https://hacks.mozilla.org/2021/02/introducing-state-partitioning/. Accessed on Apr 1 2025.

[34] David Kohlbrenner and Hovav Shacham. 2017. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*.

[35] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. 2013. Cross-origin pixel stealing: Timing attacks using CSS filters. In *CCS*.

[36] Rowan Merewood. 2019. SameSite cookies explained | Articles | web.dev. Online: https://web.dev/articles/samesite-cookies-explained. Accessed on September 10 2025.

[37] Mozilla. 2024. Introducing Total Cookie Protection in Standard Mode. Online: https://support.mozilla.org/en-US/kb/introducing-total-cookie-protection-standard-mode. Accessed on April 10 2025.

[38] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. 2011. TOTP: Time-Based One-Time Password Algorithm. RFC 6238.

[39] Marcus Niemietz and Jörg Schwenk. 2012. UI Redressing Attacks on Android Devices. *Black Hat Abu Dhabi* (2012).

[40] Jorn Nystad, Oskar Flordal, and Jeremy Davies. 2013. Methods of and apparatus for using tree representations for representing arrays of data elements for encoding and decoding data in data processing systems. US patent US8542939B2.

[41] Jorn Nystad, Oskar Flordal, Jeremy Davies, and Ola Hugosson. 2015. Methods of and apparatus for encoding and decoding data in data processing systems. US patent US9014496B2.

[42] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. 2025. Power-Related Side-Channel Attacks using the Android Sensor Framework. In *NDSS*.

[43] Sioli O'Connell, Lishay Aben Sour, Ron Magen, Daniel Genkin, Yossi Oren, Hovav Shacham, and Yuval Yarom. 2024. Pixel Thief: Exploiting SVG Filter Leakage in Firefox and Chrome. In *USENIX Security*.

[44] Gerald Palfinger, Bernd Prünster, and Dominik Julian Ziegler. 2020. AndroTIME: Identifying Timing Side Channels in the Android API. In *TrustCom*.

[45] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *CCS*.

[46] Nandika Ravi. 2024. Google Maps gets rid of another feature on Web. Online: https://tech.yahoo.com/general/articles/google-maps-gets-rid-another-220221240.html. Accessed on March 29 2025.

[47] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. 2022. Toppling top lists: evaluating the accuracy of popular website lists. In *IMC*.

[48] Paul Stone. 2013. *Pixel Perfect Timing Attacks with HTML5*. White Paper. Context Information Security. Online: https://web.archive.org/web/20130821233359/http://contextis.co.uk/files/Browser_Timing_Attacks.pdf.

[49] 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 *USENIX Security*.

[50] Güliz Seray Tuncay, Jingyu Qian, and Carl A. Gunter. 2020. See No Evil: Phishing for Permissions with False Transparency. In *USENIX Security*.

[51] Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R. Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W. Fletcher. 2024. GPU.zip: On the Side-Channel Implications of Hardware-Based Graphical Data Compression. In *S&P*.

[52] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Timing Attacks on x86. In *USENIX Security*.

[53] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. 2023. DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data. In *S&P*.

[54] John Wilander. 2020. Full Third-Party Cookie Blocking and More. Online: https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/. Accessed on April 14 2025.

[55] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *CNS*.

[56] Jincheng Yu, vvb2060, Han Wang, Howard Wu, and YuSaki Kanade. 2025. Android Hidden Api Bypass. Online: https://github.com/LSPosed/AndroidHiddenApiBypass. Accessed on April 10 2025.

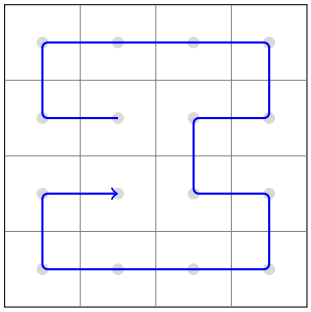


Figure 12: Order of 4×4 pixel panes within a 16×16 pixel window in Mali surface encoding.

A Pixel Enlargement Coordinate Converter

A.1 Google Pixel phones

We create a formula to calculate the offset between the coordinates used for the crop API and the ones used for the activity's window. First, we collect a set of offsets from the window's coordinates to the crop API's coordinates and observe a linear relationship in the y offset. Using linear regression, we determine the y offset to be $offset\_y = round(-0.02642857143 \cdot win\_y + 0.03571428571)$ where win_y is the window's y coordinate and $offset\_y$ is the offset from the window's y coordinate to the crop's y coordinate. The crop API's y coordinate can then be calculated by $win\_y + offset\_y$.

A.2 Samsung Galaxy S25

As in Appendix A.1, we start by collecting a set of offsets between the coordinates for the `SurfaceControl` and the window. We empirically find that the offset from the window's y coordinate to the `SurfaceControl`'s y coordinate is: $offset\_y = round(-3 * (win\_y/50) - 3)$ where win_y is the window's y coordinate and $offset\_y$ is the offset from the window's y coordinate to the `SurfaceControl`'s y coordinate. To calculate the `SurfaceControl`'s y coordinate, we can again use $win\_y + offset\_y$. We also observe that we always need to adjust the x coordinate of the window by *-1* to get the x coordinate for the `SurfaceControl`.

B Reverse Engineering the Graphical Data Compression Algorithm on Mali GPUs

We adapt Wang et al.'s `dump` utility [51] to create OpenGL textures with pixel values of our choice and save the corresponding compressed Mali surface. We then use these surfaces to reverse engineer the undocumented Mali compression format and implement a decompression utility. Our findings are summarized below.

Mali compression applies to 16×16 pixel *windows*, broken up into 4×4 pixel *panes*. In contrast to AMD and Intel, where an auxiliary surface holds compression metadata for a main surface [51], a single Mali surface comprises metadata and, afterwards, pixel data. Each window has 16 bytes of metadata: a 32-bit offset into the surface buffer where the window data are found, followed by 6 bits per pane specifying the length, in bytes, of the data for the pane. Panes are listed, in both metadata and data regions, in an unusual order that ensures that consecutive panes are contiguous (Figure 12).

As we show below, the compressed representation of a pane is at least 2 bytes long, so metadata length values 0 and 1 have special meaning: 0 that the pane has the same pixel values as the pane before it (in the order of Figure 12); 1 that the pane is stored uncompressed, as 64 bytes of RGBA pixels in row-major order.

Compressed panes use a YUVA colorspace (for decorrelation) in which the U and V channels are 9 bits, not 8. To translate from YUV to RGB, one sets $G_i \leftarrow \lfloor (515 + 4Y_i - U_i - V_i)/4 \rfloor$, $R_i \leftarrow U_i + G_i - 256$, and $B_i \leftarrow V_i + G_i - 256$.

Within a 4×4 pixel pane, each color channel is compressed separately. Let X stand for one of Y, U, V, or A.

The main idea behind Mali compression is to arrange the color values in a height-2 quadtree; child indices 0, 1, 2, and 3 correspond to directions NW, NE, SW, and SE, respectively. Let the root node be $X_\star$, the intermediate nodes be $X_{\star,i}$, and leaves be $X_{\star,i,j}$, for $i, j \in \{0, 1, 2, 3\}$. So, for example, $X_{\star,1,1}$ is the channel value for the top right pixel. Each interior node is set as the minimum of its children, i.e., $X_{\star,i} \leftarrow \min_j X_{\star,i,j}$ and $X_\star \leftarrow \min_i X_{\star,i}$. Now consider a differential representation of this tree, where nodes below the root are represented by their difference from their parent, i.e., $\Delta X_{\star,i,j} \leftarrow X_{\star,i,j} - X_{\star,i}$ and $\Delta X_{\star,i} \leftarrow X_{\star_i} - X_\star$. Observe that all ΔX values are nonnegative by construction, and that at least one child of every interior node must have $\Delta X = 0$.

The compressed representations include encodings of $X_\star$, $\Delta X_{\star,i}$, and $\Delta X_{\star,i,j}$, given which the GPU can reconstruct the channel value for each pixel.

Instead of encoding the ΔX values directly using a universal code such as exponential Golomb, Mali compression separately specifies their lengths using a separate height-1 quadtree. The length tree's root $L_\star$ specifies the encoded length of the four $\Delta X_{\star,i}$ values; for each i, the length tree leaf $L_{\star,i}$ specifies the encoded length of the four $\Delta X_{\star,i,j}$ values. As before, we use a differential representation, $\Delta L_{\star,i} \leftarrow L_{\star,i} - L_\star$. Unlike values in the X tree and its differential representation, L and ΔL values can be negative. A negative $L_{\star,i}$ means that $\Delta X_{\star,i,j} = 0$ for all j. A negative $L_\star$ means that all ΔX nodes are 0, i.e., the channel is constant; $L_\star = -2$ additionally means that $X_\star$ takes on the default channel value and is not encoded. (For the A channel, the default value is 255.)

With the notation above, we can describe the pane encoding:

- $L_\star$ for each channel, encoded as a 4-bit signed value (in two's complement).
- $\Delta L_{\star,i}$ for each channel where $L_\star \geq 0$, encoded as a 2-bit signed values (in two's complement).
- $X_\star$ for each channel where $L_\star \neq -2$, encoded as an 8- or 9-bit unsigned value depending on the channel.
- $\Delta X_{\star,i}$ for each channel where $L_\star > 0$, encoded as follows. If $L_\star$ calls for a single bit, then the value of $\Delta X_{\star,i}$ is given for each i, for a total of 4 bits. If $L_\star$ calls for more than one bit, then two bits encode the index of a zero child, i.e., i' such that $\Delta X_{\star,i'} = 0$, and then $3L_\star$ bits encode the value of $\Delta X_{\star,i}$ for $i \neq i'$, interleaved.
- Finally, for each pixel quadrant i and for each channel where $L_{\star,i} > 0$, $\Delta X_{\star,i,j}$, encoded like $\Delta X_{\star,i}$ above, either as 4 bits (if $L_{\star,i} = 1$) or as $2 + 3L_{\star,i}$ bits (if $L_{\star,i} > 1$).

The above ideas are laid out in two Arm patents [40, 41], but the implementation differs from the patents in many details.