Plain Text, Plain Risks: Measuring HTTP Inclusion in Android WebViews at Scale

Philipp Beer
TU Wien

Sebastian Roth
University of Bayreuth

Martina Lindorfer
TU Wien

Marco Squarcina
TU Wien

Abstract

While the widespread adoption of HTTPS and browser-based visual warnings for HTTP content has largely mitigated machine-in-the-middle (MitM) attacks on the traditional Web, the mobile ecosystem presents a different situation. Web content embedded via the Android WebView component commonly lacks these built-in visual security indicators and grants apps granular control over transport-layer security. This flexibility raises a critical question: does the mobile-Web ecosystem keep up with the advancements of the modern Web?

In this paper, we perform the first large-scale analysis of HTTP inclusion in WebViews across 189,779 Google Play apps. Despite Android's default policy of blocking HTTP traffic, we find that 33.74% of apps explicitly opt out. Dynamic analysis of 35,000 apps reveals that 69.96% of apps that opt out also relax the Mixed Content Policy, and we observe active HTTP traffic in 2,790. The security impact of these configurations is severe. We identify high-profile apps with 10M+ installations vulnerable to attacks ranging from phishing to full app takeover. Furthermore, we identify a major ad library transmitting cleartext ads, exposing billions of users to MitM attacks. We conclude with a qualitative developer study revealing that insecure practices are frequently driven by the requirements of third-party ad libraries and misconceptions regarding WebView's security configuration modes.

1 Introduction

Over the past decade, the widespread adoption of HTTPS has hardened the security of the Web ecosystem. Browser vendors are enforcing secure practices by visually flagging HTTP content as insecure, automatically upgrading it to HTTPS, and enforcing the Mixed Content Policy (MCP), which blocks or upgrades embedded HTTP resources by default. These efforts have made transport security a largely solved problem in the traditional Web and significantly reduced the risk posed by machine-in-the-middle (MitM) attackers, with more than 96% of web pages being loaded over HTTPS on Mozilla Firefox in North America as of the end of 2024 [36].

Meanwhile, mobile devices have become the primary medium for consuming web content [64]. Rather than relying solely on standalone browsers, many users now access web content embedded in mobile apps via so-called WebViews. These components not only allow app developers to present first-party or third-party content in a highly customizable manner, but also give developers fine-grained control over the content, such as by enabling bidirectional communication between the web content and the app. This flexibility, however, comes at the cost of potentially weakening the security of the app hosting the web content. Prior research has extensively examined the security and privacy implications of such app-to-web and web-to-app communication and uncovered numerous issues [8, 23, 40, 46, 56, 70, 75, 77]. The prevalence of loading HTTP content, a primary vector for abusing such vulnerabilities, however, remains largely unexplored.

It is tempting to assume that insights from the traditional Web automatically transfer to WebViews, yet this assumption is flawed due to fundamental differences between the mobile and traditional Web ecosystems. As WebView lacks built-in security indicators that visually flag HTTP content as "Not Secure," a standard feature in modern browsers, the risk of network attacks is obscured from users, which reduces the pressure on developers to prioritize HTTPS migration. Moreover, WebView provides developers control over programmatically downgrading web security mechanisms, such as disabling the Mixed Content Policy, and offers the ability to load local HTML data and file-based resources that blur mixed content enforcement. Most importantly, MitM attacks in WebViews extend far beyond the Web as they frequently incorporate mechanisms that connect web content to app code, allowing an attacker to escalate attacks to the app.

To understand the prevalence of these risks, we perform the first empirical large-scale analysis of HTTP content in WebViews, covering 189,779 Android apps from Google Play. By combining static and dynamic analysis, we reveal that transport security on Android remains a challenge, as 33.74% of apps explicitly opt out of Android's HTTPS-by-default policy. Dynamic analysis of 35,000 apps that opt out further

uncovers that downgrading the security guarantees of WebView is a widespread practice: 69.96% of apps allowing all HTTP traffic also disable or modify the Mixed Content Policy and 51.65% of those utilizing data-loading mechanisms use them in a way that disables mixed content enforcement, a side effect easily overlooked by developers. Consequently, we observe HTTP traffic in 2,790 apps. Crucially, our results highlight a major platform gap. 66.13% of apps loading HTTP resources remain insecure solely because WebView, unlike major modern browsers, fails to automatically upgrade HTTP connections to HTTPS, even when supported by the server.

To bridge the gap between these statistical trends and their practical security implications, we perform a security analysis of high-profile apps. We identify popular apps with 10M+ downloads that load HTTP content within their WebViews, enabling attacks ranging from UI spoofing and app interference to full app takeover. Notably, we also identify a major ad library that transmits ads over HTTP, exposing users across billions of installations to MitM attacks.

Building on these findings, we conduct a developer study as part of a large-scale responsible disclosure campaign. This campaign both informs affected developers about insecure WebView configurations and collects insights into their motivations and awareness regarding HTTP content in WebViews. Specifically, we make the following contributions:

- We compare Android WebView's handling of HTTP content with that of modern mobile browsers, revealing several security shortcomings (Section 3).
- We develop WEBVIEWTRACE, a static-dynamic analysis pipeline to systematically detect HTTP traffic and configurations in Android WebViews (Section 4).
- We execute the first empirical study of HTTP content in WebViews across 189,779 apps. We quantify a trend of security downgrades, finding that 69.96% of apps allowing all HTTP traffic also relax the MCP (Section 5).
- We illustrate the impact of our findings through case studies of popular apps, demonstrating attack vectors ranging from UI spoofing to full app takeover (Section 6).
- We conduct a responsible disclosure campaign and qualitative developer study. Engaging with 40 developers, we identify that downgrading security is caused by misconceptions, compatibility requirements, and demands from third-party SDKs (Section 7).

The artifacts of this work are publicly available at `https://doi.org/10.5281/zenodo.20393171`.

2 Background and Threat Model

In this section, we provide an overview of Android's WebView component, the ongoing efforts to transition away from plaintext HTTP traffic, and the Web's Mixed Content Policy.

2.1 Android WebView

Android developers often embed web content directly in their apps instead of launching an external browser. This approach offers a seamless user experience and enables tighter integration between the app and the web content. Most notably, apps use such components to render ads, login pages, or external links without the user having to leave the app.

One way in which Android provides this functionality is through the *WebView* [20] component. Although WebView is based on Chromium, it differs in several key aspects from a standalone browser. First, it does not come with a built-in browser interface, such as a URL bar. Second, each app's WebView instance operates in isolation, meaning that browsing data such as cookies, cache, and history are not shared with the user's default browser. And third, the WebView component exposes a rich set of APIs that allow developers to customize its behavior. For instance, apps can inject JavaScript into loaded websites via `evaluateJavascript()` or allow the website to call functions defined in the app by exposing native methods through the `addJavascriptInterface()` method [20]. Moreover, WebView allows rendering `file://` URLs and developers can specify whether files can access other files by calling `setAllowFileAccessFromFileURLs()` or whether file URLs have universal access, i.e., can access content from any origin, via `setAllowUniversalAccessFromFileURLs()` [19].

2.2 Android's HTTPS-by-Default Policy

Historically, Android permitted apps to transmit unencrypted (HTTP) traffic by default. Starting with Android 6.0 (API 23, released in 2015), developers could disable cleartext communication globally by setting the `android:usesCleartextTraffic` attribute in the app's manifest, i.e., the global app configuration file. Beginning with Android 9 (API 28, released in 2018), cleartext traffic has been disabled by default for all apps targeting this version [15]. Developers can, however, still explicitly enable cleartext communication by either setting the `usesCleartextTraffic` attribute to `true` in the manifest, or by defining a *Network Security Configuration* [17] file. The latter mechanism takes precedence over the manifest attribute and allows finer-grained control, such as restricting cleartext traffic to specific domains.

2.3 Mixed Content Policy

Even when a website is loaded over HTTPS, some of its subresources, such as images or scripts, may still be loaded over HTTP connections. Such *mixed content* undermines the integrity and confidentiality guarantees of HTTPS, as a network attacker can intercept and modify insecurely loaded resources, and thus, for instance, take control over loaded scripts.

The *Mixed Content Policy* defines a browser-enforced mechanism to mitigate these risks by restricting the loading

of HTTP resources within secure contexts. The specification [62] distinguishes between two categories: *upgradeable* (passive) mixed content and *blockable* (active) mixed content.

Upgradeable Content. This category includes low-risk resources that can be safely upgraded to HTTPS without breaking functionality, such as images, audio, or video files. When a secure page attempts to load such resources over HTTP, e.g., via the `src` attribute of `<img>`, `<audio>`, or `<video>` tags, modern browsers automatically upgrade the request to HTTPS.

Blockable Content. Blockable content refers to any type of mixed content that the browser does not consider upgradeable. This category includes high-risk resources, such as scripts, stylesheets, iframes, and fetch requests, which an attacker could exploit to compromise the security of the entire page. Browsers block this type of insecure request entirely when they originate from a secure page.

Potentially Trustworthy Origins. Requests are upgraded or blocked depending on the level of trust associated with their context, i.e., whether the request targets a *potentially trustworthy URL* and whether the embedding context originates from a *potentially trustworthy origin*. A potentially trustworthy origin is an origin that the browser can trust to deliver data securely. This includes origins served over HTTPS, localhost, and file URLs. Potentially trustworthy URLs are either URLs whose origin is potentially trustworthy or URLs that inherit the trust level of the origin they are loaded from, such as `about:blank`, `about:srcdoc`, and `data:` URLs [3, 63].

Mixed Content Policy in WebViews. WebView allows developers fine-grained control over the Mixed Content Policy. Developers can set one of the following options via the `WebSettings.setMixedContentMode()` method [19]:

- `MIXED_CONTENT_ALWAYS_ALLOW` [A]
  Allows all mixed content.
- `MIXED_CONTENT_NEVER_ALLOW` [N]
  Disallows all mixed content (default setting).
- `MIXED_CONTENT_COMPATIBILITY_MODE` [C]
  Attempts to match the behavior of modern browsers.

2.4 Threat Model

We consider a network-based *machine-in-the-middle (MitM)* attacker. The attacker is capable of intercepting and modifying traffic between an Android device and remote servers, including the manipulation of HTTP requests or responses. We assume that the attacker does not have control over the device or the app. A realistic scenario for this threat model is a user connected to a shared or untrusted network, such as a public Wi-Fi hotspot or a malicious access point.

The attacker's objective is twofold: (1) to compromise the confidentiality or integrity of the web content loaded within a WebView (e.g., via phishing or UI spoofing), or (2) to compromise the host app itself, for instance, by invoking exposed methods via injected JavaScript bridges.

3 HTTP Content Handling in WebView

While modern browsers have shifted towards strongly enforcing the Mixed Content Policy, the specific behavior of Android WebView remains less documented [19]. Thus, we conduct a systematic analysis using differential testing to compare WebView's (v142) handling of HTTP content against three mobile browsers: Google Chrome (v142), Mozilla Firefox (v147), and Brave (v1.86). We choose to test Chrome and Firefox as they represent the two major browser engines, and include Brave to evaluate the behavior of a privacy-focused browser. Furthermore, we also compare WebView v142 (released in October 2025) to WebView v124 (released in April 2024) to assess whether there have been any improvements in WebView's handling of HTTP content over time.

3.1 Methodology

We developed a test suite comprising a website served over both HTTP and HTTPS, alongside a custom Android test app. To evaluate the handling of different mixed content categories, the website embeds resources that are *upgradeable* (passive) and *blockable* (active) and serves them over HTTP. To ensure this set was representative, we curated our subresources based on the MDN classification of Mixed Content [43], including all upgradeable (e.g., images and videos) and blockable (e.g., scripts and iframes) subresources listed. The test app targets SDK 36 and runs on an Android 16 device, the latest versions as of February 2026. It is configured to permit plaintext traffic and supports all three mixed content modes: `ALWAYS_ALLOW`, `NEVER_ALLOW`, and `COMPATIBILITY_MODE`. We automated the navigation to the test site across the app and browsers and manually verified whether the content was loaded, upgraded, or blocked. In total, our test suite comprised 108 test cases.

3.2 Results

Using our test suite, we observe several differences between WebView and mobile browsers, as shown in Table 1.

WebView Versions. According to our tests, WebView v124 and v142 exhibit identical behavior across all test cases, indicating that there have been no improvements in WebView's handling of HTTP content over the past year.

Navigation. When a URL is loaded without a specified protocol (e.g., `example.com`), WebView defaults to HTTP. This behavior contrasts with all tested browsers, which automatically upgrade such requests to HTTPS. Similarly, clicking an `<a>` tag with an explicit HTTP scheme triggers an automatic upgrade to HTTPS in these browsers, whereas WebView loads the resource over HTTP. Notably, Firefox exhibits a distinct behavior where it does not upgrade these requests to HTTPS only if the target website was previously accessed via HTTP.

Mixed Content Auto-Upgrades. Chrome, Firefox, and Brave have adopted the policy of auto-upgrading upgrade-

Table 1: Treatment of HTTP resources across Chrome, Firefox, Brave, and Android WebView.

Loaded via ▶	HTTPS URL						HTTP URL				No protocol URL				File / content URL				Data with HTTPS base URL						Data with other base URL			
Browser ▶																												
MC mode ▶				[A]	[N]	[C]				[*]				[*]				[*]				[A]	[N]	[C]				[*]
Top-level load	●	●	●	●	●	●	⬆	⍟	●	○	⬆	⬆	●	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–
`a` tag navigation	⬆	⬆	⬆	○	○	○	⬆	⬆	⬆	○	⬆	⬆	⬆	○	–	–	–	○	–	–	–	○	○	○	–	–	–	○
Passive MC	⬆	⬆	⬆	○	⊘	○	⬆	⍟	⬆	○	⬆	⬆	⬆	○	–	–	–	○	–	–	–	○	⊘	○	–	–	–	○
Active MC	⊘	⊘	⊘	○	⊘	⊘	⊘	⍟	⊘	○	⊘	⍟	⊘	○	–	–	–	○	–	–	–	○	⊘	⊘	–	–	–	○

[A] `ALWAYS_ALLOW`, [N] `NEVER_ALLOW`, [C] `COMPATIBILITY_MODE`, [*] any mode, ● Served over HTTPS; ○ Served over HTTP; ⬆ Upgraded to HTTPS; ⬆ Upgraded to HTTPS (if `http://` URL not visited before); ⍟ Served over HTTP (upgraded if "HTTPS-Only" enabled); ⊘ Blocked; – not applicable.

able mixed content (e.g., images and audio) from HTTP to HTTPS. The resource is loaded if it is available over HTTPS and blocked otherwise. Our results show that WebView does not perform auto-upgrades in any mode. Notably, WebView blocks all mixed content by default (mode `NEVER_ALLOW`).

File Scheme. Although `file://` URLs should be treated as potentially trustworthy origins as per the W3C specification [63], we found they are exempt from mixed content restrictions regardless of the configured mode. Consequently, HTTP subresources loaded from a file origin are allowed to load without blocking. Notably, all tested mobile browsers lack support for loading `file://` URLs.

Data Loading APIs. Content loaded via `loadData` is assigned an opaque origin and is therefore not subject to mixed content restrictions. For `loadDataWithBaseURL`, the Mixed Content Policy enforcement depends on the base URL's scheme. We tested standard schemes (`http`, `https`, `file`, `about`, `data`, `content`, `null`, and the empty string) and determined that only URLs with `http`, `https`, `file`, and `content` schemes result in non-opaque origins. Furthermore, only data loaded with a base URL of `https` is subject to mixed content restrictions.

4 Empirical Large-Scale Evaluation

To systematically evaluate the impact of the security gaps identified in Section 3, assess the prevalence of HTTP in WebViews, and understand the rationale behind developers' decisions and their awareness of related risks, our methodology spans two main phases, as illustrated in Figure 1.

We first conduct a large-scale empirical analysis of Android apps to identify unsafe WebView configurations and HTTP traffic. To this end, we develop WebViewTrace, a static-dynamic analysis tool, and examine a broad corpus of 189,779 real-world apps. To the best of our knowledge, this is the first large-scale evaluation of HTTP traffic in WebViews. We present the results of this analysis in Section 5.

In the second phase, as described in Section 7, we perform a responsible disclosure campaign among app developers. As part of this campaign, we conduct a developer study to gain insights into developers' motivations to embed HTTP content and relax transport-layer security in WebViews. We also evaluate their awareness of the associated risks.

4.1 Dataset

To build a representative dataset of Android apps, we follow a methodology similar to Steinböck et al. [66]. We first collect a list of package names available on the Google Play Store. To this end, we crawl its public sitemap and extract all included package names, yielding 2,618,964 unique entries. For each package, we retrieve app metadata using the `google-play-scraper` [50] tool and obtain metadata for 2,485,539 apps. We then filter this set to include only free apps, resulting in a final corpus of 2,417,011 apps.

We focus primarily on popular apps, as these are more likely to be installed by users and therefore have greater implications for user security and privacy. At the same time, we aim to capture the long tail of less popular apps. To this end, we attempt to download all apps with at least 100K installs (196,362 apps) and randomly sample 5,000 apps from each of the following buckets based on the number of downloads: [0, 100), [100, 500), [500, 1K), [1K, 5K), [5K, 10K), [10K, 50K), and [50K, 100K). In total, we successfully downloaded 189,779 apps using a customized version of `apkeep` [22]. Our configuration ensures that split APKs [14] are included, which are required to execute the apps. Data collection took place between September and October 2025.

4.2 Static Analysis

We begin our analysis with a lightweight static inspection of each app's manifest and bytecode to detect apps that allow HTTP traffic and that use WebViews.

Manifest Analysis. We first decompile each APK using APKTool [28] to extract its `AndroidManifest.xml` file and parse it. Android apps can permit HTTP traffic by setting the

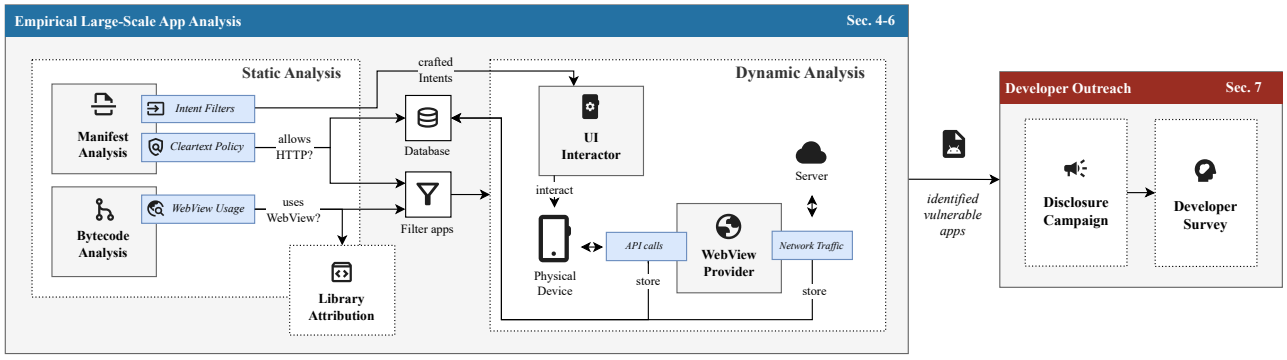


Figure 1: Our methodology consists of a large-scale measurement of 189,779 Android apps to detect unsafe WebView configurations and HTTP traffic, followed by an outreach phase comprising a developer study targeted at uncovering their root causes.

`android:usesCleartextTraffic` attribute to `true` or by defining a Network Security Configuration file. We extract both the attribute and the associated network configuration, if present, to determine whether the app allows HTTP traffic and, if so, to which domains. As the default behavior depends on the target SDK of the app, we also record this information. Beyond security configurations, we also extract all exported activities and their intent filters. We use this information during the dynamic analysis phase, where it is used to generate intents that systematically trigger app components.

Bytecode Analysis. We leverage SootUp [35], a static analysis tool that translates Dalvik bytecode into its intermediate representation (Jimple). We analyze each app's bytecode and identify whether it embeds an Android WebView by scanning for invocations of `android.webkit.WebView` APIs or any of its subclasses, and exclude apps that do not use a WebView from further analysis.

4.3 Dynamic Analysis

After we have statically analyzed each app and identified apps embedding a WebView, we perform dynamic analysis to detect unsafe WebView practices and gather run-time data. Our dynamic analysis pipeline automatically interacts with apps, triggers WebViews, and monitors their configurations and traffic. We first discuss why dynamic analysis is necessary and then describe our pipeline in detail.

4.3.1 Shortcomings of Static Analysis

We rely on dynamic analysis for three main reasons: **(1) Unreachable Code.** Apps embedding WebViews may include third-party libraries containing code that is never executed at run-time. Relying exclusively on static analysis would thus lead to an overestimation of HTTP traffic prevalence. **(2) Dynamically Loaded Content.** WebViews frequently load data that is fetched or generated dynamically, such as ads from a remote server that are subsequently served as data strings. A notable example is the Fyber SDK, which we found utilizing `loadDataWithBaseURL()` to render ads only after they are dynamically retrieved at run-time. **(3) Configuration vs. Vulnerability.** Static analysis identifies potential configuration issues, but these do not always translate into exploitable vulnerabilities. For instance, developers can implement custom interception logic via `shouldInterceptRequest()` to dynamically rewrite HTTP traffic to HTTPS, a pattern that is difficult to detect statically. Consequently, the traffic observed during dynamic analysis serves as confirmed insecure behavior.

4.3.2 UI Interactor

The UI Interactor component is responsible for simulating user interactions with the app's interface and triggering WebViews. We base our implementation on the Android Monkey tool [13], which generates pseudo-random streams of user events such as clicks, touches, or gestures to simulate user interactions with the app's interface. Despite its random exploration strategy, Monkey is used by developers and researchers and is the de facto standard UI input generation tool [4, 9].

We extend the standard Monkey tool by adding capabilities to launch specific activities based on their intent filters and thus increase activity coverage. Specifically, we construct intents for each exported activity identified during static analysis, populating the intent's data field with schemes and hosts derived from the corresponding intent filters. To target WebViews that accept URLs as launch parameters, we also generate intents that include an HTTP URL and HTTPS URL in the intent's data field, if the activity does not declare an intent filter. We prioritize activities likely to host WebViews, such as those containing "WebView" or "Browser" in their activity names, as well as those declared `BROWSABLE`.

The dynamic execution proceeds in two stages. We run the standard Android Monkey for 5 minutes to explore the UI, followed by the sequential launching of our crafted intents. The tool terminates once all intents have been issued or after a maximum run time of 15 minutes. Notably, we strictly limit

our interactions to realistic scenarios, i.e., we do not forcefully invoke non-exported activities.

Coverage. We acknowledge the inherent coverage limitations of dynamic analysis. However, while our automated tool may struggle to reach deeply nested WebViews protected by complex logic or authentication, our methodology deliberately prioritizes *breadth* to scale the analysis across a large number of apps. Furthermore, we observe that WebViews reachable with minimal interaction, such as those serving ads or managing GDPR consent, represent the apps' most immediate and frequently exposed attack surface. We evaluate our approach on a random subset of 25 apps and compare it with manual exploration, showing that our interactor, on average, covers 80.8% of the activities reached by manual analysis. A detailed analysis of activity coverage is provided in Appendix B.

4.3.3 Custom WebView Provider

To observe how apps use WebViews, we implement a custom WebView provider based on Chromium that we instrument with monitoring capabilities. Specifically, the provider records calls to relevant WebView APIs, such as `loadUrl()`, `loadDataWithBaseURL()`, and `setMixedContentMode()`, including their parameters. Furthermore, we enable WebView's network debugging interface [74] to capture resource requests and detect the outgoing traffic generated by the WebViews. As we set our provider as the system WebView, all apps requesting a WebView use our instrumented version.

4.3.4 Devices and Environment

We execute the dynamic analysis pipeline in parallel across a cluster of up to seven physical Pixel 8 devices running Android 16. All devices are connected to the internet via Wi-Fi. Each device is rooted to enable the installation of our custom WebView as the WebView provider and to grant access to the collected data files saved by the WebView.

4.4 Library Attribution

To systematically attribute WebViews to specific third-party components, we initially utilized AndroLibZoo [60]. However, we identified two critical shortcomings for our large-scale study: (1) several high-frequency libraries were absent from the dataset, and (2) it lacked essential metadata, such as the library categories and links that are important for actionable reports in our disclosure campaign. To address these gaps, we developed a hybrid attribution pipeline that complements AndroLibZoo with a custom classification based on Fully Qualified Class Names (FQCNs). We first extracted all FQCNs from call sites where WebView-related methods were invoked. If an FQCN shared the application's package name, we attributed it to first-party code. For all other instances, we extracted prefixes at a minimum depth of four segments that appeared in at least five unique apps. We leveraged an LLM (`glm-4.5-355b`) to perform the primary clustering of these candidate prefixes into distinct libraries. Specifically, we iteratively presented batches of 100 prefixes to the model to identify the corresponding library name, category, and homepage. We also queried the LLM to enrich the existing AndroLibZoo entries with metadata. To ensure precision and mitigate potential model hallucinations, we manually curated the LLM's output, resolved inconsistencies, and refined the output. For instance, we split up monolithic SDKs into distinct functional modules. Finally, we ended up with a list of 3,044 libraries. Details on the prompt are in Appendix A.

5 Large-Scale Measurement Results

We apply WEBVIEWTRACE to a corpus of 189,779 apps and quantify the real-world prevalence of insecure WebView practices at scale. We first analyze the systematic weakening of Android's Network Security Policy, which prohibits plaintext traffic by default, and attribute observed WebView usage to specific libraries. Subsequently, we evaluate the extent to which apps rely on plaintext HTTP content and which mechanisms they use, e.g., via direct navigations, data-loading APIs, or relaxing the Mixed Content Policy.

5.1 Success Rate

We executed our static analysis pipeline on the complete dataset of 189,779 apps, successfully parsed the manifest for 189,280 apps (99.74%), and performed bytecode analysis on 188,069 apps (99.10%). In total, we completed the static analysis on 187,979 apps (99.05%).

To focus our dynamic analysis on the most high-risk candidates, we filtered this dataset to include only apps that both embed an Android WebView and explicitly permit all plaintext traffic, resulting in a subset of 62,355 apps. From this group, we selected a random sample of 35,000 apps (56.13%) for dynamic exploration and successfully completed the full dynamic pipeline for 34,200 apps (97.71%).

5.2 HTTPS-By-Default Opt-Out

Starting with Android 9 (API 28), plaintext (HTTP) traffic is blocked by default unless explicitly permitted by the app's manifest or Network Security Configuration. Nevertheless, we find that a substantial number of apps override this default protection. Out of the 189,280 apps in our dataset, 63,868 (33.74%) allow HTTP traffic to all domains, and 84,032 apps (44.40%) allow it to at least one domain.

Allowed Domains. The most commonly allowed domains correspond to local addresses (`*.127.0.0.1` in 19,148 apps, `*.localhost` in 2,419 apps) or emulator hostnames (`*.10.0.2.2` in 1,340 apps), likely used for local debugging. Beyond these cases, we identify Amazon's ad system

(`*.amazon-adsystem.com` in 1,434 apps) and Tencent's Bugly crash-reporting endpoint (`*.android.bugly.qq.com` in 472 apps) [69]. Furthermore, we notice several appearances of different content delivery and ad networks.

Distribution of Opt-In. Opting in to HTTP traffic is not restricted to less-used or legacy apps. As can be seen in Figure 2, both apps with over 1M installations and apps released across the years, based on when the app first appeared on the Google Play Store, often completely disable Android's HTTPS-by-default policy. Interestingly, while the number of apps that allow all traffic is consistent across installations, the number of apps disallowing all traffic rises. The "Unknown" category refers to apps where the static analysis was not able to infer the cleartext policy, e.g., when the Network Security Configuration file could not be found.

Configuration Issues. Our analysis revealed 42 apps with invalid configurations in the `cleartextTrafficPermitted` attribute. We empirically confirmed that the parser is highly permissive: any value not strictly representing `false` (including `FALSE` and `0`) is interpreted as `true`. Beyond simple typographical errors, we identified cases where developers used non-boolean strings such as `trust` or relied on Manifest placeholders (e.g., `${ipHittingEnabled}`), which are not resolved.

Adoption over Time. Our results show that, by the end of 2025, 55.60% of apps prevent cleartext traffic to all domains, compared to 33% found in a prior study by Possemato et al. [53] in 2020. Notably, however, 39.4% of apps in the 2020 dataset targeted API levels $\leq$ 27 (prior to the HTTPS-by-default policy), whereas only 0.07% of our dataset targets these legacy versions. Furthermore, while 33.3% of apps in the 2020 study explicitly declared a Network Security Policy (whether allowing or disallowing cleartext), this metric rises to 50.30% in our dataset. This demonstrates that, despite the long-standing enforcement of secure defaults, the ecosystem has not fully converged, as developers continue to explicitly opt out of the HTTPS-by-default policy.

5.3 WebViews Explored

We statically identified WebViews being used in the vast majority of apps. Specifically, we found references to WebView APIs in 183,017 apps (97.31%). Our dynamic instrumentor triggered WebView APIs in 25,186 apps (73.64%) and WebView instances in 23,237 apps (67.94%) out of 34,200 successfully dynamically analyzed apps that allow HTTP traffic to all domains. Note that some WebView APIs are not associated with instances. For example, an app may call `CookieManager.getCookie()` to retrieve cookies, but never initialize a WebView instance.

Unique WebViews. Among apps allowing all HTTP traffic, we trigger an average of 5.18 unique WebViews within a given app with a median of 4.00. We define a unique WebView as the invocations of key configuration settings, e.g., whether

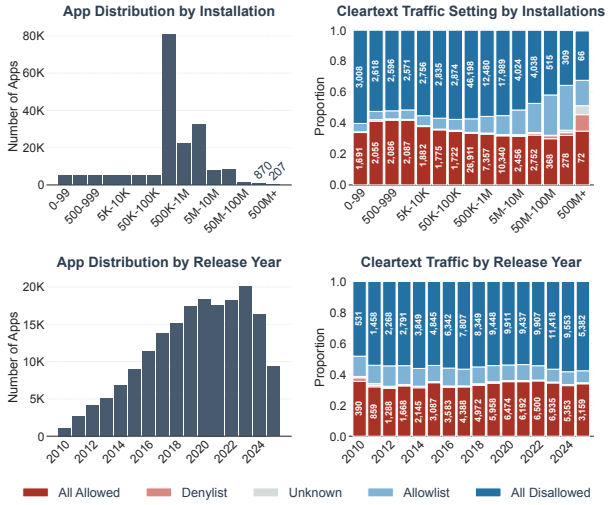


Figure 2: Distribution of apps' cleartext traffic settings.

JavaScript is enabled, the specific Mixed Content Policy settings, and exposed JavaScript bridges. To distinguish multiple WebView instances across app restarts, we also consider the originating call stack up to a depth of five. Calls to methods such as `loadUrl()` are treated equivalently regardless of their parameters. This prevents overcounting when the UI automator re-triggers the same WebView across app restarts.

Origins of WebViews. Most WebViews triggered in the set of apps allowing all HTTP traffic originate from libraries rather than the app's own code. Among our dataset, most WebViews triggered originate from the Google Mobile Ads SDK, followed by the Google User Messaging Platform. Notably, only 3.46% of WebViews in this set of apps are loaded from first-party code. We also observe instances where a WebView is "shared" between libraries, i.e., calls from multiple libraries occur on the same WebView instance. Specifically, this is the case in 6.98% of WebViews. Most commonly among apps, a WebView is shared between Google Mobile Ads and SafeDK in 5,462 apps, followed by IronSource and SafeDK in 4,337 apps, and Unity and SafeDK in 4,199 apps. SafeDK [1] is a third-party library monitoring SDK targeted at monitoring ad visibility and brand safety and is now incorporated into AppLovin [26]. We furthermore observe that the most common categories of libraries triggered include advertising libraries, followed by libraries that handle GDPR consent. We note that this might be due to our exploration strategy. Dynamic analysis struggles to reach code paths that are deep in the application and favors code paths that can be executed without substantial interaction. Advertising libraries, such as Google Mobile Ads, or consent libraries are frequently loaded early on, which is why our analysis skews towards these. Figure 3 shows the top 10 sources of WebViews that we triggered. "Unknown Source" refers to cases where we were unable to attribute a WebView to a specific library or first-party code.

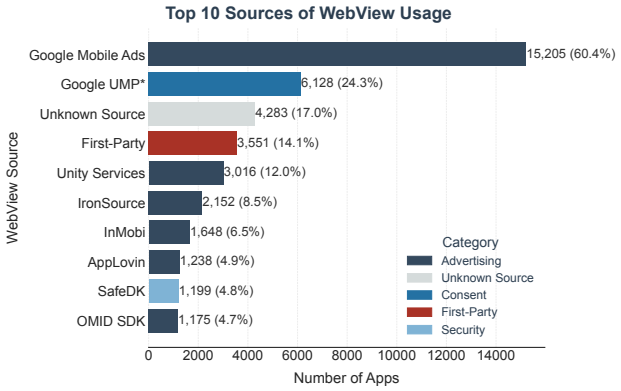


Figure 3: WebView Usage in Android apps. Google UMP refers to the Google User Messaging Platform. Percentages relate to the set of dynamically analyzed apps using WebViews and allowing all HTTP traffic ($N = 25,186$).

Scheme	# Apps	Scheme	# Apps
`https`	19,563 (77.67%)	`http` (external)	1,354 (5.38%)
`about:blank`	10,796 (42.87%)	`http` (local)	925 (3.67%)
`file`	4,399 (17.47%)	`empty string`	835 (3.32%)
`javascript`	4,075 (16.18%)	`https` (local)	445 (1.77%)

Table 2: Top 8 URL schemes loaded via `loadUrl()`. Local refers to `localhost`, `127.0.0.1`, and `0.0.0.0` ($N = 25,186$).

5.4 Insecure `loadUrl` Navigations

One way HTTP-based content can be loaded into WebViews is when an app directly loads an HTTP URL using the `loadUrl()` API. As established in Section 3, loading an HTTP URL results in an automatic upgrade to HTTPS in Chrome, Brave, and Firefox (in HTTPS-Only Mode). WebView, however, does not perform automatic upgrades for such navigations, nor for navigations where no scheme is provided.

Usage and Loaded Schemes. Out of 25,186 apps that allow all HTTP traffic and where we dynamically detected Android WebViews, 21,831 apps (86.68%) call the `loadUrl()` API. We find that the most commonly used scheme is `https`, followed by `about:blank`, `file`, `javascript`, and `http`. While loading `about:blank` is used to clear the WebView, `javascript` can be used to execute JavaScript code in the context of the website. Notably, 8.97% of these apps request an HTTP URL, whereas 5.38% load a non-localhost HTTP URL. Table 2 shows the top 8 URL schemes loaded by the number of apps.

Missing Protocol. We identified 67 apps that load a non-empty URL lacking a protocol scheme. As discussed, for these URLs, WebView defaults to using HTTP. However, only 21 apps request a domain for which a DNS lookup succeeded, while the remaining apps loaded malformed URLs.

Local Address HTTP URLs. In total, 925 apps load local addresses using `loadUrl()`. The primary source of local address-based HTTP requests is the StartApp SDK, an advertising library that loads `http://0.0.0.0` in 608 apps. While heavy obfuscation prevents us from pinpointing the exact reason, we note that the URL resolves to `localhost`. Beyond this specific SDK, local HTTP traffic is predominantly generated by hybrid frameworks, specifically Ionic with Capacitor [30, 31] and the Ionic Cordova WebView plugin [32]. Ionic/Capacitor and the Ionic Cordova WebView plugin use the HTTP URL only as a placeholder to serve local content and override the `shouldInterceptRequest()` method to directly return the local content. While no network requests are made for these URLs, the choice of protocol significantly impacts the app's overall security. For instance, Capacitor 5 (and earlier) serves the app via an HTTP URL [33]. Because the top-level origin is considered insecure, the WebView does not enforce the Mixed Content Policy, regardless of the mixed content configuration, implicitly allowing the loading of external HTTP subresources. Conversely, Capacitor 6 (released in 2024) migrated to HTTPS, ensuring that mixed content is blocked by default unless explicitly overridden. The Ionic Cordova WebView plugin still serves the app via HTTP by default.

External HTTP URLs. We observe 827 HTTP domains loaded via `loadUrl()`. 13.30% of these domains are used by more than one app, and 10.52% are used by more than one developer, demonstrating that the majority of loaded HTTP domains are specific to an app, rather than originating from generic third-party services.

5.5 Mixed Content Enforcement Gaps

When data is loaded via `loadData()`, a unique mechanism of WebViews, or files are loaded via `loadUrl()`, the Mixed Content Policy is not enforced, regardless of the mixed content configuration. Consequently, HTTP subresources are neither upgraded to HTTPS nor blocked. This also applies to `loadDataWithBaseURL()` with a non-HTTPS base URL.

API Usage. We find that APIs to load data and files are commonly used: 3,325 apps (13.20% of those allowing all cleartext traffic and using WebViews) load data via `loadData()`, 26 apps via `loadUrl()`, and 13,008 apps (51.65%) via `loadDataWithBaseURL()`. Files are loaded in 4,397 apps.

Commonly Used Base URLs. The WebView guide notes that the URL used as the base URL "must be an HTTP(S) URL" [16]. Nevertheless, we find that a large number of apps use non-HTTP(S) base URLs, and these undermine the security provided by the Mixed Content Policy. Notably, we find that while 10,140 apps load data with an HTTPS-based URL, 6,705 use `about:blank`, 3,460 a `file` URL, and 1,863 use `null`. In total, 51.65% of apps using a WebView and allowing all cleartext traffic use a non-HTTPS URL.

5.6 Systematic Policy Weakening

By default, Android WebView blocks the loading of mixed content (i.e., HTTP content loaded within an HTTPS page).

	Source	Active MC	Passive MC
Ad	Google Mobile Ads	0 (0.00%)	15,164 (60.21%)
–	First-party	609 (2.42%)	85 (0.34%)
Ad	Fyber SDK	0 (0.00%)	564 (2.24%)
Ad	Mintegral	549 (2.18%)	0 (0.00%)
🎓	Classplus	533 (2.12%)	0 (0.00%)
📱	Ionic Cordova WebView	203 (0.81%)	0 (0.00%)
–	Google Mobile Services	82 (0.33%)	4 (0.02%)
📱	AgentWeb	68 (0.27%)	0 (0.00%)
📱	Enhanced WebView	67 (0.27%)	19 (0.08%)
📱	Flutter InAppWebView	21 (0.08%)	60 (0.24%)

Table 3: Top 10 sources allowing mixed content, sorted by total apps (Ad advertising, 🎓 e-learning, 📱 UI component). Percentages relate to the set of analyzed apps using WebViews and allowing all HTTP traffic ($N = 25,186$).

However, apps can modify this behavior by calling the `setMixedContentMode()` method of the `WebSettings` class to allow all mixed content or apply the compatibility mode that, as we have discussed in Section 3, allows passive mixed content but blocks active mixed content.

Prevalence. The usage of the `setMixedContentMode()` method among apps is ubiquitous. We record 18,204 apps (72.28% of those that use WebViews and allow all HTTP traffic) calling this method. Furthermore, 3,312 apps (13.15%) allow all mixed content (A), including scripts, iframes, and stylesheets, 15,578 apps (61.85%) only allow passive mixed content via the compatibility mode (C), and 17,620 (69.96%) allow either passive or all mixed content (A or C).

Sources. Table 3 details the top 10 sources responsible for relaxing the Mixed Content Policy. The landscape is dominated by ad SDKs, such as Google Mobile Ads [25], Mintegral [67], and Fyber [21]. Specifically, the Google Mobile Ads SDK accounts for the vast majority of policy relaxations. While heavy obfuscation prevents us from pinpointing the exact internal rationale, the Google Ad documentation notes that while HTTPS is now the default for app ad serving in most countries, "there are a few exceptions where HTTPS is optional." Moreover, the documentation notes that the "Ad Manager scans creatives for HTTPS compliance. If a creative is not HTTPS-compliant (e.g., it has mixed content with some HTTP references), it should still fully display on the app, but there will be an entry in the internal logs" [24]. Beyond advertising, we identify specific platforms driving insecure defaults. Classplus, an ed-tech platform allowing educators to build branded apps, enables active mixed content for most of its WebViews. Similarly, hybrid frameworks and UI wrappers frequently downgrade security. The Ionic Cordova WebView plugin [32] defaults to allowing all mixed content (`ALWAYS_ALLOW`), while the popular Enhanced WebView [12] plugin defaults to compatibility mode.

Recorded Mixed Content. We dynamically captured confirmed instances of HTTP resources loaded in an HTTPS context across 182 apps. We emphasize that this figure represents a lower bound. Our dynamic analysis prioritizes app UI exploration rather than performing deep crawling of the loaded web content itself. Thus, mixed content that is located deeper within the loaded websites likely remains unobserved.

5.7 WebView Capabilities

Beyond merely displaying visual content, WebViews can expose other sensitive functionality. For example, apps can explicitly enable JavaScript execution and expose native methods to web scripts via the `addJavascriptInterface()` method. Furthermore, developers can grant local files access to other files using `setAllowFileAccessFromFileURLs()` or allow them to bypass the same-origin policy entirely via `setAllowUniversalAccessFromFileURLs()`. If a network attacker successfully injects malicious scripts into such a WebView, they can abuse these elevated privileges.

To evaluate the potential impact of MitM attacks exploiting these capabilities, we analyze their prevalence across the WebView instances in our dataset. Specifically, we examine whether WebViews with exposed capabilities systematically downgrade the Mixed Content Policy and whether the resources they load are subject to Mixed Content Policy enforcement in the first place. WebView instances that explicitly permit active mixed content, or those that load resources exempt from the Mixed Content Policy, fail to provide any meaningful defense against MitM attacks leveraging HTTP-based subresources. Table 4 summarizes the distribution of these capabilities with respect to their mixed content configurations.

Evaluation Criteria. We define a WebView as loading content subject to the Mixed Content Policy only if it invokes `loadUrl()` or `loadDataWithBaseURL()` using an HTTPS scheme. Conversely, loading resources via any other scheme indicates that the content is exempt from MCP enforcement; naturally, a single WebView instance may load both types of content. In cases where we observe multiple conflicting calls to `setMixedContentMode()`, we conservatively assume the safest configuration, i.e., that the WebView blocks mixed content. This ambiguity affects 0.52% of all observed instances. Similarly, if we encounter conflicting capabilities (e.g., calling `setJavaScriptEnabled()` with both `true` and `false` in an instance), we assume the more restrictive state. We denote these conflicting cases with a dagger in Table 4.

Risk. Our results highlight severe risks: 99.1% of WebView instances that permit active mixed content also enable JavaScript, and 67.1% expose a JavaScript bridge. Even among the 38.8% of WebViews configured to block mixed content, i.e., a setting that seemingly mitigates HTTP-subresource-based MitM attacks, we find that a majority (50.4%) load content exempt from MCP enforcement. Furthermore, 7.3% of all WebView instances load local files, which inherently bypasses the MCP, and crucially, 52.7% of these instances allow scripts to bypass the same-origin policy.

MC Configuration / Loaded Content	WebView Instances									
	JS		JS Bridge		File access		Universal access		Total	
Allows active MC	24,094†	99.1%	16,327	67.1%	3,800*	15.6%	5,859*	24.1%	24,323	3.7%
Loads content subject to MCP	10,548†	98.8%	7,759	72.7%	874*	8.2%	1,045*	9.8%	10,679	1.6%
Loads content not subject to MCP	10,929†	98.7%	7,695	69.5%	2,555*	23.1%	4,051*	36.6%	11,069	1.7%
Allows passive MC	375,213*	99.1%	374,668	98.9%	564*	0.1%	641	0.2%	378,794	57.5%
Loads content subject to MCP	317,594*	100.0%	317,146	99.8%	254	0.1%	262	0.1%	317,670	48.2%
Loads content not subject to MCP	74,953†	95.8%	76,116	97.3%	205	0.3%	272	0.3%	78,237	11.9%
Blocks MC	200,120†	78.2%	96,964	37.9%	25,007*	9.8%	28,194*	11.0%	255,777	38.8%
Loads content subject to MCP	46,675†	86.4%	24,546	45.4%	1,683†	3.1%	2,180†	4.0%	54,012	8.2%
Loads content not subject to MCP	128,885†	90.5%	53,789	37.8%	22,794*	16.0%	25,160*	17.7%	142,450	21.6%
Loads `file://`	45,882†	95.7%	36,397	75.9%	22,272*	46.4%	25,280*	52.7%	47,958	7.3%
Total	599,427†	91.0%	487,959	74.1%	29,371*	4.5%	34,694*	5.3%	658,894	100.0%

Table 4: WebViews by mixed content configuration (JS enabled; JS Bridge exposed; File access from file URLs; Universal access from file URLs). Asterisks and daggers denote instances with conflicting settings (*: $< 0.5\%$ of instances affected, †: $< 2.5\%$ of instances affected).

5.8 Loaded Content

In total, we observe HTTP traffic leaving the WebView over the network interface in 2,790 apps (11.08% of those using WebViews and allowing HTTP traffic to all domains).

Resource Types. Table 5 breaks down loaded HTTP resources by initiator scheme and whether the resource was loaded due to systematic Mixed Content Policy relaxation or mixed content enforcement gaps. We determine resource types primarily via URL file extensions and fall back to Content-Type sniffing when no extension is available. Top-level navigations represent the most frequent source of HTTP traffic (8.2% of apps). Specifically, 1,944 apps directly initiated a navigation to an HTTP page, while in 129 apps, a navigation to such a page was initiated by an HTTPS-served page (e.g., via an `<a>` tag navigation). While 529 apps load HTTP-served scripts, the majority are embedded in HTTP-served pages. Only 8 apps loaded an HTTP script due to mixed content relaxation. Moreover, 129 apps initiated an HTTP resource from a `null` scheme, which, for example, is used when the main page is loaded via `loadData()`. Finally, we identified HTTP traffic stemming from Authority Information Access (AIA) Fetching [10] in 80 apps, i.e., the process of downloading missing intermediate SSL/TLS certificates. We exclude URLs ending with `.crt` from further analysis.

Auto-Upgrade. We evaluate the feasibility of auto-upgrading these unsafe resources. We identify 29,960 unique external URLs requested over HTTP. During our follow-up testing, a subset of 4,318 URLs (14.41%) was no longer reachable via either HTTP or HTTPS. Among the remaining URLs, we find that 20,951 (81.71%) were accessible over HTTPS. In total, all HTTP URLs could be upgraded to HTTPS without breaking functionality in 66.13% of apps. These results demonstrate that the majority of insecurely loaded resources are already served via HTTPS. However, as our analysis in Section 3 shows, Android WebView, unlike Chrome, Firefox, and Brave, fails to perform auto-upgrades for these resources. Given that a large portion of requested content is also served over HTTPS, implementing a platform-level auto-upgrade mechanism for WebViews would significantly mitigate the risks without compromising content availability.

6 Case Studies

While the measurement results in Section 5 demonstrate that insecure WebView configurations are prevalent, in this section, we showcase their concrete impact through case studies of high-profile apps affecting millions of users.

6.1 UI Spoofing

HTTP content embedded in WebViews fundamentally undermines the visual trust of the hosting app. When content is loaded over plaintext HTTP, MitM attackers can intercept and modify the response to perform *UI spoofing*. This enables phishing attacks where attackers can, for example, inject prompts instructing users to perform sensitive actions.

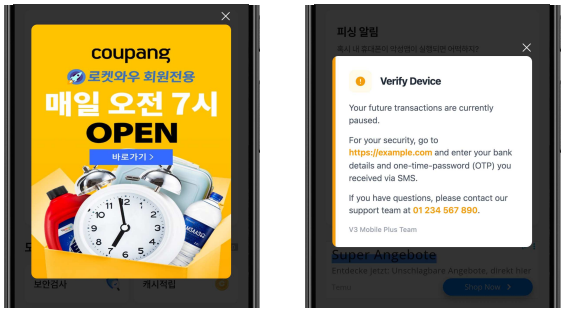
This risk is significantly exacerbated in WebViews compared to standard browsers. Unlike browsers, WebViews typically lack visual security indicators, such as a URL bar or a lock icon indicating that the content was transmitted over HTTPS. Consequently, users generally have no visual means to distinguish between a legitimate app interface and malicious injected content, making them more susceptible to social engineering attempts compared to standard browsers. We demonstrate the practicality and impact of this attack through two case studies, MapFactor and V3 Mobile Plus.

Loaded Resource	Initiator scheme: HTTPS	HTTP	File	Browser-initiated	`null`	Total
Top-Level Page	**129 (0.5%)** ⊙	87 (0.3%)	11 (<0.1%)	**1,944 (7.7%)** ⊙	3 (<0.1%)	2,057 (8.2%)
Script	**8 (<0.1%)** ↓	482 (1.9%)	**25 (<0.1%)** ⊙	–	**23 (<0.1%)** ⊙	529 (2.1%)
Subframe	**2 (<0.1%)** ↓	63 (0.3%)	**19 (<0.1%)** ⊙	–	**3 (<0.1%)** ⊙	87 (0.3%)
Font	**7 (<0.1%)** ↓	139 (0.6%)	**10 (<0.1%)** ⊙	–	**5 (<0.1%)** ⊙	159 (0.6%)
CSS	**6 (<0.1%)** ↓	441 (1.8%)	**10 (<0.1%)** ⊙	–	**5 (<0.1%)** ⊙	453 (1.8%)
Image	**90 (0.4%)** ↓	1,259 (5.0%)	**42 (0.2%)** ⊙	–	**83 (0.3%)** ⊙	1,428 (5.7%)
Audio	**3 (<0.1%)** ↓	16 (<0.1%)	**1 (<0.1%)** ⊙	–	**12 (<0.1%)** ⊙	32 (0.1%)
Video	**2 (<0.1%)** ↓	11 (<0.1%)	–	–	**1 (<0.1%)** ⊙	14 (<0.1%)
Certificate	–	–	–	80 (0.3%)	–	80 (0.3%)
Others*	4 (<0.1%) ↓	86 (0.3%)	15 (<0.1%) ⊙	–	–	102 (0.4%)
Unknown	56 (0.2%)	235 (0.9%)	16 (<0.1%)	6 (<0.1%)	5 (<0.1%)	315 (1.3%)
Total	**299 (1.2%)**	**1,340 (5.3%)**	**97 (0.4%)**	**2,005 (8.0%)**	**129 (0.5%)**	**2,790 (11.1%)**

Table 5: Apps loading HTTP resources by type and initiator (↓ resource loads that result from systematic MCP weakening; ⊙ resource loads that originate from MC enforcement gaps compared to Chrome). *Others* refers to HTML, JSON, Text, XML, and PDF. Percentages relate to the set of analyzed apps using WebViews and allowing all HTTP traffic ($N = 25,186$).

MapFactor. *MapFactor* (`com.mapfactor.navigator`) is a navigation app with 10M+ downloads. The app displays custom ads within a WebView. We observed that the ad content is served as a top-level HTML page over HTTP from `http://static.mapfactor.com`. A MitM attacker can modify the content to perform phishing attacks by replacing the ad with what appears to be an app prompt. The attacker can configure the content so that any click on the prompt leads the user to an attacker-controlled website, where the attack can proceed. Notably, as the ad served by MapFactor is also accessible via HTTPS, the vulnerability could have been prevented if the WebView component performed auto-upgrades to HTTPS like Chrome or Firefox.

V3 Mobile Plus. An example of the risk introduced by relaxing the Mixed Content Policy is given by *AhnLab V3 Mobile Plus* (`com.ahnlab.v3mobileplus`), a mobile security app with 50M+ downloads. Although the app's main screen is served via HTTPS, it loads subresources, specifically the pop-up image shown at launch, over HTTP (Figure 4a). The WebView is explicitly configured with `COMPATIBILITY_MODE`, allowing these insecure subresources to render. This allows an attacker to replace the image with a fraudulent instruction, such as a "Verify Device" alert prompting the user to call a support number, as shown in Figure 4b. Similar to the previous case study, the resource is also available over HTTPS, yet the Android WebView does not perform an auto-upgrade.



(a) Real pop-up. (b) Manipulated pop-up.

Figure 4: *V3 Mobile Plus*: Pop-up on app launch.

6.2 App Interference

Beyond phishing, insecure WebViews risk compromising the app itself. As established in Section 5.7, if an app exposes a JavaScript bridge via the `addJavascriptInterface()` method and the page loads scripts using HTTP, a MitM attacker can invoke these methods. Similarly, apps can also use a custom scheme to handle communication with the app to expose such capabilities to the Web. Depending on the exposed functionality, this can lead to severe consequences ranging from PII exfiltration to the execution of app-defined actions, as the following case studies demonstrate.

11st. *11st* (`com.elevenst`) is a major online marketplace in South Korea with over 10M app downloads. We observed that important app configurations, such as image URLs, or the URL of the login website, are loaded via HTTP. Upon launch of the app, a MitM attacker can force them to load over HTTP, and, for example, change the login URL to an attacker-controlled login page. Crucially, the WebView used for login exposes six high-privilege JavaScript bridges, including `AndroidIDCardOCR` and `AndroidSKPaySDK`. These interfaces expose sensitive functions such as `saveUserInfo`, `showMoneyCharge`, `requestIDCardRecognize`, and `showToast`. We initially verified the exploitability of these interfaces by performing a call to `AndroidSKPaySDK.showToast`, which triggers a toast with attacker-controlled content within the app.

Fyber SDK. The *DT Exchange SDK* (formerly Fyber) is a widely used ad exchange platform. Our analysis reveals that this SDK loads ad creatives using the `loadDataWithBaseURL()` method, with the base URL dynamically determined by the app's network policy. Crucially, if the host app permits cleartext traffic, the SDK sets the base URL to an `http://` scheme. This behavior effectively bypasses mixed content protections entirely. Because the WebView perceives the ad content as originating from an insecure HTTP context, it permits the loading of arbitrary active content (scripts) and passive content (images) regardless of the WebView's mixed content mode. We observed ads served through networks like Liftoff, Adform, and Remerge leveraging this insecure context to load HTTP JavaScript resources. While Fyber's documentation [21] notes a "Secure Only Mode" to allow apps to only load from secure sources, it is disabled by default. The security implications extend beyond visual spoofing or phishing. Mobile ads typically adhere to the MRAID (Mobile Rich Media Ad Interface Definitions) standard [29], which grants ads access to device capabilities. Assuming MRAID capabilities, an MitM attacker can inject malicious JavaScript into the HTTP traffic to invoke sensitive MRAID 2.0 functions, such as `storePicture` (writing files to storage) or `createCalendarEvent`. We observed HTTP traffic originating from this library, i.e., HTTP requests initiated by `http://wv.inner-active.mobi`, in 290 apps.

6.3 App Takeover

Hybrid app frameworks like Apache Cordova implement the application logic within a WebView, which requires exposing sensitive JavaScript bridges. Consequently, embedding HTTP content within these apps carries significant risk. Crucially, Cordova apps are often served from the `file://` origin. As highlighted in Section 5.7, allowing `file://` schemes to bypass the same-origin policy (SOP) creates a critical security failure: if an attacker achieves code execution (e.g., by modifying a script loaded over HTTP), the code inherits `file://` privileges and can access content from any origin.

GLOBE Observer. The NASA GLOBE Observer app (`gov.nasa.globe.observer`), a Cordova-based app with 100K+ downloads, executes in a WebView via the `file` scheme. We discovered that the app fetches a JavaScript resource via HTTP, which allows a MitM attacker to run code in the `file://` context. Because the app utilizes Cordova's JavaScript bridges and explicitly invokes `setAllowUniversalAccessFromFileURLs()`, the injected script gains access to sensitive device permissions (e.g., the camera) and can bypass the SOP to exfiltrate data from any origin within the WebView.

7 Understanding Developer Practices

Building on our large-scale analysis, we conducted a qualitative survey through a responsible disclosure campaign to identify the root causes of unsafe WebView practices and assess developer awareness of the associated risks.

7.1 Study Design and Methodology

Recruitment & Disclosure. We contacted developers of Android apps for which we identified unsafe WebView practices. We obtained their contact information from the publicly available email addresses listed in the developers' Google Play Store entries, as several previous works incorporating responsible disclosure campaigns relied upon [47–49, 61]. Each invitation email contained a brief description of our study, a preview of the issues that we identified, and a link to our project's website including a developer report and the survey.

Report. We generated a report summarizing the issues detected for each developer's apps. Developers could view the report without completing the survey, although we encouraged them to complete it before accessing the report.

Survey. The survey consisted of 5 blocks, containing 2-5 multiple-choice or free-text questions. We began with questions about the developer's background and Android development experience. We then asked whether they were aware that their apps used a WebView and for what purpose. Subsequent questions focused on the usage of specific WebView configurations and their awareness of their security implications. Participants could skip any question. The interested reader is referred to the paper's artifacts for the complete survey.

Data Analysis. To analyze the free-text responses, we conducted an inductive thematic analysis following the methodology of Braun and Clarke [5], a common approach in security-focused user and developer studies [6, 34, 59]. Initially, one author familiarized themselves with the data and generated a codebook. Based on this codebook, two authors independently coded the data and subsequently resolved any existing conflicts through discussion.

7.2 Engagement and Demographics

We sent out reports and invitations to the survey to 6,892 developers[1]. 193 developers accessed our website, leading to an engagement rate of 2.80%. 44 developers (0.64%) viewed their report, 48 (0.70%) accessed the survey, and 40 started it.

The cohort was geographically diverse, spanning 21 countries across Europe, Asia, Africa, and the Americas. Participants were highly experienced: 14 reported over 10 years of professional experience, and only 6 reported fewer than 5 years. Roles were predominantly technical, including 23 developers and 4 security engineers.

[1] In total, we sent out disclosures to 11,116 developers; however, 4,224 disclosures were distributed after our cutoff date and were thus excluded from further analysis.

7.3 Developer Awareness and Understanding

We find that while developers generally grasp high-level security concepts, their understanding of WebView-specific behavior is often superficial.

7.3.1 WebView Usage

The motivations for embedding WebViews varied significantly among participants. The most frequent use cases included displaying external websites (16), rendering legal documents such as Terms of Services (9), and serving ads (8). 4 participants reported using a hybrid framework, which relies on WebViews for the core app UI. One participant noted that they are more familiar with web development than app development and therefore relied on web technologies instead.

Notably, while most practitioners (23) were aware of WebView usage within their apps, 5 respondents were either unaware that the component was being used or explicitly said that they do not use WebViews. 12 did not provide an answer.

7.3.2 Awareness of Security Policies

Although all contacted developers implemented a network policy, awareness of this mechanism was mixed. Among the 23 respondents, only 17 (74%) were aware they implemented such a policy. When queried about its function, most participants (14) correctly identified that it controls cleartext traffic. Only P37 was unaware of its purpose, and P35 incorrectly assumed it was also required to load file URLs in WebViews.

7.3.3 Configuration Misconceptions

The Mixed Content Policy was less understood than the Network Security Configuration. Four of 15 respondents explicitly stated they did not know what the setting does. Among those familiar with the concept, while the general idea of blocking HTTP subresources was clear, specific behaviors were often unclear, particularly regarding the compatibility mode. P35 admitted that they did not *"really know the policy that make android choose to block a content or let it go,"* while P12 mentioned that *"some dynamic content will fail to load."* Notably, we also observed misconceptions in how the Mixed Content Policy is enforced. For instance, P2 incorrectly noted that the compatibility mode automatically blocks all mixed content for modern target SDKs (however, it allows passive content irrespective of the target SDK), while P23 erroneously believed that `NEVER_ALLOW` prevents HTTP loading within `loadData()`. Interestingly, three participants expressed apathy about these configurations.

7.3.4 Risks of HTTP Inclusion

When asked about the risks of HTTP usage, the cohort demonstrated a strong bias towards confidentiality concerns. While the vast majority mentioned MitM risks as potential implications, the security impacts they mentioned largely focused on data leakage (11). In contrast, WebView-specific integrity risks, such as app interference by abusing JavaScript bridges, were largely overlooked. Only 3 participants mentioned that an attacker can interfere with the app. Similarly, no respondent mentioned the possibility of local file access or same-origin policy bypasses. This suggests that while developers understand that HTTP is unencrypted and allows attackers to eavesdrop and tamper with the content, they do not always have WebView-specific implications in mind. To mitigate the issue, our participants suggested security indicators similar to those in browsers. Specifically, P23 mentioned that *"users have no browser security indicators and may fully trust the in-app content"* and P35 suggested to *"clearly notify the user that the website opened could be harmful"*.

7.4 Root Causes of Insecure Configurations

Our analysis reveals that insecure configurations are rarely the result of isolated negligence. Instead, they often emerge as necessary trade-offs driven by external constraints or operational requirements.

7.4.1 Requirement from Third-Party Libraries

Third-party libraries, especially ad SDKs, are a dominant driver of policy degradation. Here we see parallels to the web ecosystem, where third parties are a frequently mentioned reason for lowering the security boundaries offered by CSP or Trusted Types [57, 58, 65]. Four developers explicitly stated they allowed HTTP traffic or relaxed mixed content modes because an ad provider mandated it. Two developers specifically mentioned the Appodeal SDK, and our manual verification confirmed that the Appodeal documentation requires allowing all HTTP traffic [2]. P25 explained that *"almost all ad networks want you to configure it that way,"* likely because *"they still have some customers advertising* `http://` *websites."*

7.4.2 Development Artifacts

Three participants relaxed transport-layer protections to facilitate local development. P11 admitted they simply *"forgot to remove that settings from the release."* Notably, although Android provides the `<debug-overrides>` mechanism to handle such cases, these developers appeared unaware of the feature.

7.4.3 Legacy Support and Local Web Services

Participants also mentioned the need to allow users to browse any website through their built-in browser (2), relying on local services (2) and external services/websites they have no control over (3). Two participants stated that they are currently in the process of transitioning from HTTP to HTTPS, while one mentioned that they rely on a legacy app code base.

7.4.4 Problems with SSL Certificates

Uniquely, P15 offered a rationale grounded in geopolitical risk. They deliberately allow cleartext traffic to mitigate the risk of service interruption due to international sanctions: *"there is no guarantee that services will be provided properly if your origin is questioned by certain states, imposing restrictive sanctions."* Furthermore, they also mentioned financial investment for certificates as a reason to allow HTTP traffic.

7.4.5 Misconceptions

Finally, we observed that security is sometimes traded for user experience. While P2 selected `COMPATIBILITY_MODE` due to a misunderstanding (believing it was safe), P14 correctly understood that it permits passive content but justified its use to *"ensure a consistent user experience without breaking the visual layout",* prioritizing the rendering of mixed content images over strict security enforcement. Moreover, one participant mentioned relying on an app template and not configuring the transport-layer security themselves.

8 Limitations

UI Exploration Coverage. Our dynamic analysis relies on automated UI exploration, which inherently suffers from incomplete code coverage. WebViews protected by authentication, complex navigation paths, or rare event triggers may remain unexplored. However, we emphasize that the "shallow" WebViews captured by our analysis represent the most immediate attack surface, as these components are exposed without requiring authentication or complex interaction sequences.

Developer-Study Bias. Our study relies on self-selected participants, introducing selection bias inherent to notification campaigns. While respondents typically exhibit higher security awareness than the general population, this bias reinforces our findings regarding the Android ecosystem. Even within our cohort, we identified developers who were unaware of their app's WebView usage and configurations. This suggests that awareness among the broader developer community is likely significantly lower.

9 Related Work

WebView and Browser Security. Extensive research has examined WebView security, particularly focusing on JavaScript bridges, which enable communication between the app and the web content [8, 23, 40, 46, 56, 76] and resource manipulation within WebViews [77]. Weerasekara et al. [75] studied how WebView's bidirectional communication channels are exploited to track users and circumvent platform privacy protections. Nguyen et al. [49] analyzed WebView's permission system, identifying thousands of apps that automatically grant all requested permissions to embedded WebViews. Zhang et al. [78] studied the UI of in-app browsers, showing that many fail to indicate connection security or origin clearly to users. Other research [7, 71, 79] explored the emerging *app-in-app* ecosystem and revealed a range of privacy and privilege separation issues. Tiwari et al. [70] performed a static analysis of Android-Web hybridization, primarily focusing on insecure information flows between native code and JavaScript bridges. While they statically identified the presence of unencrypted HTTP schemes in `loadUrl()` calls, their study did not investigate the run-time behavior of these connections. Mutchler et al. [45] inspected the problem of target SDK fragmentation on Android and also considered unsafe WebView defaults, such as `setAllowFileAccessFromFileURLs()` and `setMixedContentMode()`. However, their analysis was performed purely statically and did not go beyond reporting general trends nor did it focus on HTTP inclusion. Oltrogge et al. [51] performed a static and dynamic analysis of apps created by 13 online application generators. While they also analyzed mixed content settings in WebViews, their study was restricted to the niche domain of automatically generated apps, rather than the broader Android ecosystem. Datta et al. [11] investigated tracking on mobile devices based on app-to-web interactions in Android Apps. To do so, they modified an app-crawling framework, created custom Frida gadgets responsible for hooking WebView-based Java functions inside Android apps, and modified VisibleV8 to run inside the rendering process of an app-loaded WebView by compiling their own version of a system-wide VisibleV8 WebView provider. We also based our analysis on replacing the system's WebView provider. However, instead of hooking into a VisibleV8 Chromium via Frida, we directly added logging to the Android WebView APIs in order to not suffer from any problems or limitations of Frida. Concerning mobile browser security, Luo et al. [39] analyzed the support of web security mechanisms among mobile browsers. In our work, we do not focus on a specific security mechanism, but rather on the inclusion of HTTP resources within a WebView, the different behavior of WebViews compared to browsers, and the security issues that arise from this behavior. Furthermore, Pradeep et al. [55] analyzed mobile browsers, finding several security and privacy shortcomings, such as not validating TLS certificates.

HTTP and TLS/SSL Issues on Android. Prior work has analyzed Android's network security policies. Oltrogge et al. [52] evaluated the Network Security Configuration across 100k apps, revealing that 89% of apps with an NSC effectively downgrade their network-layer security. Similarly, Possemato et al. [53] analyzed NSC adoption and found that a majority of apps explicitly permit cleartext traffic, often driven by ad libraries or insecure code snippets from StackOverflow. Their study relied on static analysis and did not investigate mixed content or dynamically measure the actual prevalence of plaintext traffic in the wild. More recently, Moon et al. [44] performed static analysis on 25 apps to detect improper TLS

implementations, including those in WebViews. Pourali et al. [54] developed a dynamic analysis tool to detect certificate validation vulnerabilities, finding insecure configurations in 6.4% of Google Play Store apps compared to 55.3% in Chinese app stores. Zuo et al. [80] and Liu et al. [38] combined static and dynamic analyses to show that around 5% of Android apps override WebView's `onReceivedSslError` handler and ignore SSL errors. Wang et al. [72, 73] introduced a tool to detect certificate validation vulnerabilities in both app and WebView code based on static and dynamic analysis. Most recently, Lee et al. [37] analyzed the certificate validation mechanism of in-app browsers among 20 popular apps.

Large-Scale Responsible Disclosure. Nguyen et al. have conducted multiple large-scale responsible disclosure campaigns targeting Android developers, examining GDPR violations [47], tracking consent [48], and the WebView permission system [49]. Response rates in these studies varied significantly, ranging from 9.3% to 17% for report access and roughly 1% to 3.7% for survey participation. Most recently, Schmidt et al. [61] contacted 661 developers regarding exposed secrets, receiving replies from 37 (5.6%). Regarding disclosure optimization, Maass et al. [41] found that adding detailed attack scenarios did not improve remediation rates for emails and negatively impacted them for letters. Complementary studies have established best practices for disclosing issues to website owners [42] and designing trustworthy security notifications [27] to maximize engagement.

10 Conclusion

In this paper, we presented the first large-scale analysis of HTTP inclusion in Android WebViews. While the traditional Web has successfully transitioned to HTTPS, our results demonstrate that these security guarantees often fail when content is embedded within mobile apps.

Our analysis of 189,779 apps reveals that the "secure-by-default" mechanism of modern Android versions is frequently undermined in practice. We found that 33.74% of apps explicitly opt out of Android's HTTP traffic blocking, and dynamic analysis of 35,000 apps that use WebViews and opt out showed that 69.96% actively weaken the Mixed Content Policy. We further observe that the widespread use of data-loading APIs like `loadData()` and `loadDataWithBaseURL()` frequently assigns content to non-secure origins, effectively bypassing mixed content restrictions and allowing insecure subresources to load unchecked. Consequently, we observed HTTP traffic in 11.08% of apps using WebViews and allowing all HTTP traffic, exposing millions of users to MitM attacks.

Crucially, our findings indicate that insecure configurations are often driven by the requirements of major third-party advertising SDKs and hybrid frameworks, rather than developer negligence alone. Furthermore, as our qualitative study shows, developers are often unaware of their security configurations and their specific behavior. This risk is exacerbated by the fact that WebViews lag behind modern browsers: All HTTP URLs in 66.13% of the vulnerable apps we identified could have been upgraded automatically if Android WebView adopted the auto-upgrade mechanisms of modern browsers.

To address these issues, we recommend that Android align WebView's handling of mixed content and HTTP navigations with the stricter standards of modern mobile browsers. Furthermore, acknowledging that WebViews currently lack the visual cues found in browsers, we echo the suggestion of our survey participant to implement clear security indicators for insecure in-app content to alert users to potential risks. To assist developers in assessing whether their app relies on HTTP traffic, including within WebViews, we recommend the use of a `VmPolicy` [18] with `detectCleartextNetwork`. This policy can be configured to either log violations or crash the process, providing immediate visibility into insecure traffic.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and the participants of our study for their time and effort. This work was supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060], the Austrian Science Fund (FWF) [Grant ID: 10.55776/F8515-N], Cyber Security Austria (CSA), and SBA Research (SBA-K1 NGC), a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG. The paper includes icons from Font Awesome and Material Icons and the analysis scripts made use of GNU Parallel [68], a parallel execution tool.

Open Science

To facilitate future research and ensure the reproducibility of our findings, we publish the following artifacts.

HTTP Behavior Comparison. The source code, including the Android app and website server, for the HTTP behavior comparison discussed in Section 3.

Static Analysis. The source code and per-app results of the static analysis phase comprising the manifest analysis and the bytecode analysis introduced in Section 4.2.

Dynamic Analysis. The source code and per-app results of the dynamic UI instrumenter and the custom WebView provider presented in Section 4.3.

Developer Survey. The full questionnaire used in our developer survey described in Section 7.

These artifacts are publicly available at `https://doi.org/10.5281/zenodo.20393171`. Due to storage constraints, we

cannot publicly redistribute the APK dataset. However, the full APK dataset is available upon request.

Ethical Considerations

Stakeholder Analysis. Our work involves the following four stakeholder groups: (i) *app store operators and OS providers (i.e., Google)*, (ii) *app developers*, (iii) *end-users*, and (iv) *security researchers*. In the following, we detail how our methodology mitigates potential harms and impacts each group.

App Dataset Download. To ensure minimal impact on the Google Play Store's infrastructure, i.e., stakeholder (i), we implemented strict rate limiting during our data collection phase (September-October 2025). We throttled the process to a maximum of four parallel downloads with a 7-second cool-off period between each request. Because we do not publicly redistribute the downloaded APKs, we prevent copyright impacts on stakeholder (ii). As we make the dataset available to researchers upon request, we believe we are facilitating future research and thus positively impacting stakeholder (iv).

Automated UI Exploration. We utilized a modified version of the Android Monkey tool for automated UI exploration. To mitigate potential side effects for app developers, i.e., stakeholder (ii), and their backend systems, we took several precautions: (1) we conducted all experiments on dedicated research devices containing no personal user data to prevent information leakage, (2) we throttled our interactions to better imitate real user interaction, and (3) we did not perform dedicated logins, avoiding any accidental modification of real user accounts and thus protecting app users, i.e., stakeholder (iii).

Responsible Disclosure. Our study involved a large-scale responsible disclosure campaign, notifying all affected app developers, i.e., stakeholder (ii), of the specific security flaws identified within their apps. These notifications provided actionable insights, including mitigation recommendations and technical documentation, in the developers' best interests. We have received multiple responses from vendors thanking us for our insights and stating they have fixed the issues, e.g., by moving from network-fetched assets to local assets. Our disclosure ultimately protects end-users, i.e., stakeholder group (iii), from MitM risks. Finally, we have also informed the Android Security Team (i) about our findings and the lack of default HTTP-to-HTTPS auto-upgrades in WebViews. They flagged the report as "working as intended" and logged the issue for potential remediation in a future version.

Developer Survey. During our responsible disclosure campaign, we also invited affected developers, i.e., stakeholder group (ii), to participate in our survey. Participation in the associated survey was entirely voluntary, and participants could access their specific vulnerability report regardless of whether they chose to participate. We performed data minimization where applicable, followed our institution's best practices for human-centered studies (e.g., providing participants with an information sheet listing their rights and potential risks for participation) and the questions were strictly limited to professional development practices and decision-making. Specifically, we did not collect sensitive information about the participants.

Impact of Publication. We recognize that the publication of this research may negatively impact the reputation of stakeholder (ii). Furthermore, as app vendors may have not fully mitigated all risks in the affected apps until publication, there is a risk that malicious actors could use our findings to target users, i.e., stakeholder group (iii), until mitigations are deployed. However, we believe that the benefits of publication, i.e., raising awareness of HTTP usage in WebViews among researchers (iv) and developers (ii), and informing app developers about security issues within their apps, outweigh the risk of potential misuse by malicious actors, particularly as our responsible disclosure provided a timeline for developers to fix these flaws prior to public release, a point underlined by multiple app developers thanking us for reaching out to them.

References

[1] AppLovin. SafeDK – Manage your SDKS. https://web.archive.org/web/20200207080600/https://www.safedk.com/, February 2020. Last accessed: February 2, 2026.

[2] Appodeal Inc. Get Started. https://docs.appodeal.com/android/get-started, February 2026. Last accessed: February 5, 2026.

[3] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web Platform Threats: Automated Detection of Web Security Issues With WPT. In *USENIX*, 2024.

[4] Jakob Bleier, Felix Kehrer, Jürgen Cito, and Martina Lindorfer. Profile Coverage: Using Android Compilation Profiles to Evaluate Dynamic Testing. In *ASE*. IEEE/ACM, 2025.

[5] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3, 2006.

[6] Annalina Buckmann, Jan Magnus Nold, Yasemin Acar, and Yixin Zou. More than Usability: Differential Access to Digital Security and Privacy. In *SOUPS*, 2025.

[7] Yifeng Cai, Ziqi Zhang, Mengyu Yao, Junlin Liu, Xiaoke Zhao, Xinyi Fu, Ruoyu Li, Zhe Liu, Xiangqun Chen, Yao Guo, et al. I Can Tell Your Secrets: Inferring Privacy Attributes from Mini-app Interaction History in Super-apps. In *USENIX*, 2025.

[8] Erika Chin and David Wagner. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *WISA*. Springer, 2013.

[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated Test Input Generation for Android: Are We There Yet? In *ASE*. IEEE/ACM, 2015.

[10] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://www.rfc-editor.org/info/rfc5280, May 2008.

[11] Sohom Datta, Michalis Diamantaris, Ahsan Zafar, Junhua Su, Anupam Das, Jason Polakis, and Alexandros Kapravelos. Cross-Boundary Mobile Tracking: Exploring Java-to-JavaScript Information Diffusion in WebViews. In *NDSS*, 2026.

[12] delight-im. Android-AdvancedWebview. https://github.com/delight-im/Android-AdvancedWebView, 2025. Last accessed: February 1, 2026.

[13] Android Developers. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey, April 2023. Last accessed: January 13, 2026.

[14] Android Developers. About Android App Bundles. https://developer.android.com/guide/app-bundle, December 2025. Last accessed: January 13, 2026.

[15] Android Developers. <application>. https://developer.android.com/guide/topics/manifest/application-element, June 2025. Last accessed: October 21, 2025.

[16] Android Developers. Load in-app content. https://developer.android.com/develop/ui/views/layout/webapps/load-local-content, December 2025. Last accessed: January 12, 2026.

[17] Android Developers. Network security configuration. https://developer.android.com/privacy-and-security/security-config, September 2025. Last accessed: October 21, 2025.

[18] Android Developers. StrictMode.VmPolicy.Builder. https://developer.android.com/reference/android/os/StrictMode.VmPolicy.Builder.html, March 2025. Last accessed: January 29, 2026.

[19] Android Developers. WebSettings. https://developer.android.com/reference/android/webkit/WebSettings, April 2025. Last accessed: January 12, 2026.

[20] Android Developers. WebView. https://developer.android.com/reference/android/webkit/WebView, September 2025. Last accessed: January 31, 2026.

[21] Digital Turbine. Integrating the Android SDK. https://developer.digitalturbine.com/hc/en-us/articles/360010822437-Integrating-the-Android-SDK, November 2025. Last accessed: October 28, 2025.

[22] EFForg. apkeep. https://github.com/EFForg/apkeep, 2025. Last accessed: October 6, 2025.

[23] Mohamed A El-Zawawy, Eleonora Losiouk, and Mauro Conti. Vulnerabilities in Android webview objects: Still not the end! *Computers & Security*, 109, 2021.

[24] Google Ad Manager Help. Serve ads on apps securely over HTTPS. https://support.google.com/admanager/answer/6118579?hl=en, 2026. Last accessed: January 23, 2026.

[25] Google Developers. Google Mobile Ads SDK. https://developers.google.com/admob/android/sdk, February 2026. Last accessed: February 1, 2026.

[26] Anthony Ha. AppLovin acquires SafeDK to improve brand safety. https://techcrunch.com/2019/07/09/applovin-acquires-safedk-to-improve-brand-safety/, July 2019. Last accessed: February 2, 2026.

[27] Anne Hennig, Fabian Neusser, Aleksandra Alicja Pawelek, Dominik Herrmann, and Peter Mayer. Standing out among the daily spam: How to catch website owners' attention by means of vulnerability notifications. In *CHI*, 2022.

[28] iBotPeaches. Apktool. https://github.com/ibotpeaches/apktool, 2025. Last accessed: November 4, 2025.

[29] Interactive Advertising Bureau. Mobile Rich-media Ad Interface Definitions (MRAID) v.2.0. https://iabtechlab.com/wp-content/uploads/2016/02/IAB_MRAID_v2_FINAL.pdf, April 2013. Last accessed: February 2, 2026.

[30] Ionic. Capacitor. https://capacitorjs.com/, 2025. Last accessed: January 21, 2026.

[31] Ionic. Ionic Framework. https://ionicframework.com/, 2025. Last accessed: January 21, 2026.

[32] Ionic Team. cordova-plugin-ionic-webview. https://github.com/ionic-team/cordova-plugin-ionic-webview, 2025. Last accessed: January 23, 2026.

[33] Dallas James. Capacitor Android customScheme issue with Chrome 117. https://ionic.io/blog/capacitor-android-customscheme-issue-with-chrome-117, October 2023. Last accessed: January 21, 2026.

[34] Kelechi G Kalu, Tanmay Singla, Chinenye Okafor, Santiago Torres-Arias, and James C Davis. An Industry Interview Study of Software Signing for Supply Chain Security. In *USENIX*, 2025.

[35] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. SootUp: A Redesign of the Soot Static Analysis Framework. In *TACAS*. Springer, 2024.

[36] Christoph Kerschbaumer, Frederik Braun, Simon Friedberger, and Malte Jürgens. The State of https Adoption on the Web. In *MADWeb*, 2025.

[37] Woonghee Lee, Junbeom Hur, and Hyunsoo Kwon. Deep Dive into In-app Browsers: Uncovering Hidden Pitfalls in Certificate Validation. In *CCS*. ACM, 2025.

[38] Yang Liu, Chaoshun Zuo, Zonghua Zhang, Shanqing Guo, and Xinshun Xu. An automatically vetting mechanism for SSL error-handling vulnerability in android hybrid Web apps. *World Wide Web*, 21, 2018.

[39] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *NDSS*, 2019.

[40] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android system. In *ACSAC*, 2011.

[41] Max Maass, Marc-Pascal Clement, and Matthias Hollick. Snail Mail Beats Email Any Day: On Effective Operator Security Notifications in the Internet. In *ARES*, 2021.

[42] Max Maaß, Henning Pridöhl, Dominik Herrmann, and Matthias Hollick. Best Practices for Notification Studies for Security and Privacy Issues on the Internet. In *ARES*, 2021.

[43] MDN Web Docs. Mixed Content. https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Mixed_content, November 2025. Last accessed: May 15, 2026.

[44] Iffath Tanjim Moon, Afsana Mimi, and Md. Musfiqur Rahman Mridha. Improper Implementation of Transport Layer Security: Impact on Android Applications and Man-in-the-Middle Attack. In *PEEIACON*, 2024.

[45] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. Target Fragmentation in Android Apps. In *SPW*. IEEE, 2016.

[46] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A View to a Kill: WebView Exploitation. In *LEET*, 2013.

[47] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share First, Ask Later (or Never?) Studying Violations of GDPR's Explicit Consent in Android Apps. In *USENIX*, 2021.

[48] Trung Tin Nguyen, Michael Backes, and Ben Stock. Freely Given Consent? Studying Consent Notice of Third-Party Tracking and Its Violations of GDPR in Android Apps. In *CCS*. ACM, 2022.

[49] Trung Tin Nguyen and Ben Stock. Open Access Alert: Studying the Privacy Risks in Android WebView's Web Permission Enforcement. In *AsiaCCS*. ACM, 2025.

[50] Facundo Olano. google-play-scraper. https://github.com/facundoolano/google-play-scraper, 2025. Last accessed: October 6, 2025.

[51] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *S&P*. IEEE, 2018.

[52] Marten Oltrogge, Nicolas Huaman, Sabrina Klivan, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In *USENIX*, 2021.

[53] Andrea Possemato and Yanick Fratantonio. Towards HTTPS Everywhere on Android: We Are Not There Yet. In *USENIX*, 2020.

[54] Sajjad Pourali, Xiufen Yu, Lianying Zhao, Mohammad Mannan, and Amr Youssef. Racing for TLS Certificate Validation: A Hijacker's Guide to the Android TLS Galaxy. In *USENIX*, 2024.

[55] Amogh Pradeep, Álvaro Feal, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, and David Choffnes. Not Your Average App: A Large-scale Privacy Analysis of Android Browsers. In *PETS*, 2023.

[56] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In *RAID*. ACM, 2018.

[57] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In *CCS*. ACM, 2021.

[58] Sebastian Roth, Lea Gröber, Philipp Baus, Katharina Krombholz, and Ben Stock. Trust Me If You Can - How Usable Is Trusted Types In Practice? In *USENIX*, 2024.

[59] Aakanksha Saha, James Mattei, Jorge Blasco, Lorenzo Cavallaro, Daniel Votipka, and Martina Lindorfer. Expert Insights into Advanced Persistent Threats: Analysis, Attribution, and Challenges. In *USENIX*, 2025.

[60] Jordan Samhi, Tegawendé F. Bissyandé, and Jacques Klein. AndroLibZoo: A Reliable Dataset of Libraries Based on Software Dependency Analysis. In *MSR*. ACM, 2024.

[61] David Schmidt, Sebastian Schrittwieser, and Edgar Weippl. Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps. In *CCS*. ACM, 2025.

[62] Emily Stark, Mike West, and Carlos Ibarra Lopez. Mixed Content. https://www.w3.org/TR/mixed-content, February 2023. Last accessed: March 26, 2025.

[63] Emily Stark, Mike West, and Carlos Ibarra Lopez. Secure Contexts. https://www.w3.org/TR/secure-contexts/, November 2023. Last accessed: October 21, 2025.

[64] Statista. Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 2nd quarter 2025. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices, September 2025. Last accessed: October 5, 2025.

[65] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In *NDSS*, 2021.

[66] Magdalena Steinböck, Jakob Bleier, Mikka Rainer, Tobias Urban, Christine Utz, and Martina Lindorfer. Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses. In *MSR*. ACM, 2024.

[67] Jeff Sue. Getting Started with Mintegral's Open-Source SDK. https://www.mintegral.com/en/blog/getting-started-with-the-mintegral-opensource-sdk, September 2020. Last accessed: February 1, 2026.

[68] Ole Tange. GNU Parallel, November 2023.

[69] Tencent. Bugly. https://bugly.qq.com/v2/. Last accessed: October 28, 2025.

[70] Abhishek Tiwari, Jyoti Prakash, Sascha Gross, and Christian Hammer. A Large Scale Analysis of Android — Web Hybridization. *Journal of Systems and Software*, 170, 2020.

[71] Mona Wang, Pellaeon Lin, Jeffrey Knockel, Will Greenberg, Jonathan Mayer, and Prateek Mittal. What WeChat Knows: Pervasive First-Party Tracking in a Billion-User Super-App Ecosystem. In *PETS*, 2025.

[72] Yingjie Wang, Xing Liu, Weixuan Mao, and Wei Wang. DCDroid: Automated Detection of SSL/TLS Certificate Verification Vulnerabilities in Android Apps. In *TURC*. ACM, 2019.

[73] Yingjie Wang, Guangquan Xu, Xing Liu, Weixuan Mao, Chengxiang Si, Witold Pedrycz, and Wei Wang. Identifying vulnerabilities of SSL/TLS certificate verification in Android apps with static and dynamic analysis. *Journal of Systems and Software*, 167, 2020.

[74] WebView Docs. Net debugging in WebView. https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/net-debugging.md, July 2024. Last accessed: January 13, 2026.

[75] Nipuna Weerasekara, José Miguel Moreno, Srdjan Matic, Joel Reardon, Juan Tapiador, Narseo Vallina-Rodríguez, et al. Tracking Without Borders: Studying the Role of WebViews in Bridging Mobile and Web Tracking. In *PETS*, 2025.

[76] Zhanhui Yuan, Zhi Yang, Jinglei Tan, and Hongqi Zhang. WebViewJSdetect: JavaScript Vulnerability Detection in Android WebView via Coverage-Guided Thread-Adaptive Concurrent Abstract Interpretation. *Computer Networks*, 275, 2025.

[77] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, XiaoFeng Wang, Long Lu, and Haixin Duan. An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications. In *USENIX*, 2018.

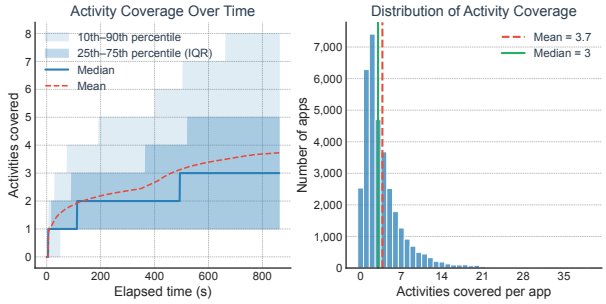
[78] Zicheng Zhang. On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps. In *RAID*. ACM, 2021.

[79] Zidong Zhang, Qinsheng Hou, Lingyun Ying, Wenrui Diao, Yacong Gu, Rui Li, Shanqing Guo, and Haixin Duan. MiniCAT: Understanding and Detecting Cross-Page Request Forgery Vulnerabilities in Mini-Programs. In *CCS*. ACM, 2024.

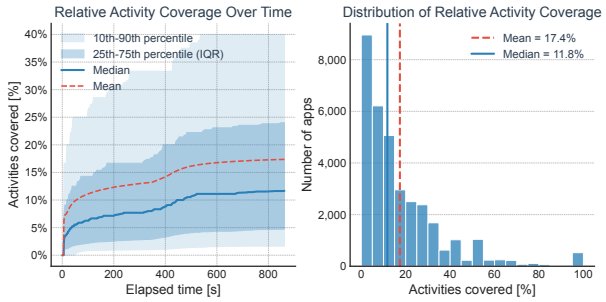
[80] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps. In *AsiaCCS*. ACM, 2015.

A Library Classification

Listing 1 shows the prompt used for library classification.



(a) Absolute activity coverage.



(b) Relative activity coverage.

Figure 5: Activity coverage of the UI Interactor across all dynamically analyzed apps.

```
You are analyzing Android libraries.
You are given a list of classes which you should
     attempt to group into libraries.
The list of classes is given below.

CLASSES:
{classes}

Respond with a structured JSON object containing
     an array of objects. Each object must
    include:
- 'name': the library name,
- 'prefix': the library prefix,
- 'category': the library type (e.g., Analytics,
     Advertising, UI Components, Networking)
- 'link': (optional) a URL to the library's
    homepage or repository if available.
```

Listing 1: Library classification prompt.

B UI Interactor Coverage

In this section, we provide insights into the dynamic coverage.

Complete Dataset. Across all apps dynamically analyzed, we achieved an avg. activity coverage of 3.7 activities (17.4%). Figure 5 shows the activity coverage across all apps over time.

App	Login	Duration	Act. Count I	Act. Count M	Act. Coverage I	Act. Coverage M	C_I/C_M
1*	–	125s	1	1	7.1%	7.1%	100.0%
2	●	44s	4	3	2.8%	2.1%	133.3%
3		220s	1	1	33.3%	33.3%	100.0%
4	●	100s	8	4	3.5%	1.7%	200.0%
5	○	157s	11	8	5.3%	3.9%	137.5%
6		904s	3	8	3.6%	9.6%	37.5%
7		904s	2	8	3.5%	14.0%	25.0%
8		904s	3	15	10.7%	53.6%	20.0%
9	●	100s	1	1	10.0%	10.0%	100.0%
10		903s	3	2	16.7%	11.1%	150.0%
11		904s	5	5	4.6%	4.6%	100.0%
12		175s	5	7	3.1%	4.4%	71.4%
13†	–	100s	1	1	9.1%	9.1%	100.0%
14		412s	4	3	3.5%	2.6%	133.3%
15		290s	4	5	33.3%	41.7%	80.0%
16		64s	4	4	66.7%	66.7%	100.0%
17		214s	8	6	66.7%	50.0%	133.3%
18	●	102s	4	2	15.4%	7.7%	200.0%
19	○	903s	4	9	19.0%	42.9%	44.4%
20	○	181s	4	4	12.5%	12.5%	100.0%
21		250s	5	9	33.3%	60.0%	55.6%
22	○	904s	11	31	10.9%	30.7%	35.5%
23‡	–	65s	2	1	1.9%	0.9%	200.0%
24	○	714s	3	2	27.3%	18.2%	150.0%
25	●	43s	2	3	5.6%	8.3%	66.7%
Mean			**4.1**	5.7	**16.4%**	20.3%	**80.8%**§
Median			**4.0**	4.0	**10.0%**	10.0%	**100.0%**

* During manual interaction, the app fails with a request timeout error.
† During manual interaction, the app requires an update and redirects to the browser for this update. The app is not available for download.
‡ During manual interaction, the app requires an update to launch. Even after update, the app fails with "Disconnected".
§ Calculated as the ratio of means ($\text{Mean}_{C_I}/\text{Mean}_{C_M}$) rather than the mean of individual ratios.

Table 6: Evaluation of the UI Interactor (I) compared to a human baseline (M) across 25 randomly sampled apps. *Login* indicates authentication in the app (● requires authentication; ○ authentication available), *Duration* refers to the duration of the manual interaction, and $\mathbf{C_I/C_M}$ represents the activity coverage of the Interactor (C_I) relative to the human baseline.

Comparison to Human Baseline. To assess the effectiveness of our UI Interactor, we compare its performance against a human baseline. We sampled 25 apps from our dataset and manually interacted with them. Because the UI Interactor (as described in Section 4.3.2) is set up to run for a maximum of 15 minutes, we mirrored this approach and capped manual interaction at 15 minutes per app. If we covered all UI screens before the time limit, we stopped the interaction. Furthermore, we did not attempt to perform authentication for apps that require it; however, we note whether an app offers or requires authentication to progress further. Table 6 summarizes the results across all tested apps. On average, the UI Interactor reaches 80.8% of the activities covered by the human baseline per app. However, the UI Interactor strictly outperforms the human baseline in 9 out of 25 apps. The interested reader is referred to the paper's artifacts for a mapping of app IDs in Table 6 to package names.