Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Bachelorthesis

Byetrack: Capabilities as a Solution against Tracking Across Android Apps

submitted by

Tim Christmann

on November 20, 2025

Reviewers

Dr. Sven Bugiel

Prof. Dr. Andreas Zeller

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich für folgende zulässige Teilaufgaben: **code generation, literature research, text rewriting/revision**, und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

Declaration of Original Authorship

I hereby declare that this thesis is my own original work and was completed independently, without unauthorized assistance or unacknowledged sources. Any content derived from publications or other external sources, whether quoted verbatim or paraphrased, has been duly acknowledged and clearly marked as such. I hereby confirm that, in preparing the following thesis, I have used artificial intelligence-based software (such as ChatGPT) solely for the following permitted tasks: **code generation, literature research, text rewriting/revision**, and not to address the core research questions presented in this thesis. All parts of the thesis produced or modified with the assistance of artificial intelligence-based software (such as ChatGPT) have been clearly indicated as such and that the software used has been listed as a resource. I acknowledge that any breach of this declaration may result in failing the examination and, in severe cases, the right to be examined may be revoked.

Ort/Place, Datum/Date	Unterschrift/Signature

Abstract

Trusted Web Activities (TWAs) and Custom Tabs (CTs) enable Android developers to seamlessly integrate web content into native applications, offering a powerful tool for features such as Single Sign-On and in-app monetization. However, as demonstrated by HyTrack, this integration also introduces severe privacy risks by blurring the boundary between web and app contexts, allowing tracking through the browser's shared cookie storage.

In this work, we propose Byetrack, a novel mitigation framework that applies capability-based access control to browser cookie handling. Cookie access is encapsulated in fine-grained, identity-bound capabilities, ensuring that only trusted first-party or explicitly authorized third-party web servers — defined by a developer-controlled policy — can access the shared browser state. All other untrusted third-party domains are confined to isolated, app-local cookie jars. Importantly, these isolated cookie jars are not only used internally to confine untrusted domains but are also exposed as a stable, developer-usable storage mechanism for domains that are intentionally scoped to the application.

Byetrack is designed to be fully backwards compatible with legacy applications: apps that do not provide a policy automatically retain the standard CT/TWA behavior without requiring any modifications. At the same time, developers who opt in gain precise control over how cookie state is shared across the app–web boundary, enabling improved privacy guarantees without requiring changes to existing web content. Our approach thus balances privacy and usability, enabling tracking-resistant web–app integration while preserving essential features such as Single Sign-On, personalized content delivery, and compatibility with existing Android applications.

Acknowledgements

I would like to thank Noah Mauthe for supervising this thesis and for his continuous guidance and support. His constructive feedback and our many discussions were invaluable in shaping the ideas and implementation in this work.

I also thank Dr. Sven Bugiel for giving me the opportunity to carry out this thesis in his group and for his helpful input throughout the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, Android applications have increasingly integrated web content into their interfaces to enhance user experience and streamline features such as authentication and monetization. To enable this, developers commonly rely on Custom Tabs (CTs) [2] and Trusted Web Activities (TWAs) [3], technologies that allow seamless, browser-backed web integration while preserving native-like performance and appearance. This integration enables web-based functionality such as Single Sign-On (SSO) or in-app advertising without forcing users to leave the application or manage separate browser sessions.

However, these benefits come at a cost. CTs and TWAs share the browser's cookie storage across all apps, providing session continuity but also introducing severe privacy vulnerabilities. Recent research by Wessels et al. [1] demonstrated HyTrack, a novel tracking technique that exploits this shared browser state to persistently identify users across different applications and the web, even surviving device changes, cookie clearing, or browser switching.

HyTrack enables persistent cross-app tracking as long as developers integrate the tracking library as instructed. Its identifiers can even survive device resets through Android's backup mechanisms, giving it Evercookie-like persistence [4].

In this thesis, we present Byetrack — to the best of our knowledge, the first mitigation framework that addresses these privacy issues while preserving the legitimate use cases of CTs and TWAs, despite the HyTrack authors' claim that this would require major changes to their underlying design. Byetrack allows developers to preserve the benefits of CTs and TWAs, such as SSO and monetization, while preventing invisible cross-app tracking identified by HyTrack. Our approach applies the principle of capability-based security: fine-grained, unforgeable tokens that grant specific access rights to a resource. In this context, the browser issues capabilities based on a developer-defined policy, which

explicitly determines which domains may access the shared browser state. All other domains are confined to app-local storage, thereby preventing unauthorized cross-app cookie sharing while preserving trusted integrations.

Beyond mitigating HyTrack-style tracking, Byetrack gives developers the option to fully decouple their web content from the shared browser state, allowing independent cookie management for greater control and privacy.

Our evaluation demonstrates that Byetrack effectively prevents HyTrack's cross-app tracking vector while preserving Android's core functionality and existing web features. On the app side, developers only need to adopt our Byetrack-enabled browser library and provide a small policy file; the remaining support is supplied by a slightly modified Android version.

Chapter 2

Background

Before presenting the design and implementation of our mitigation, we first provide the necessary background on tracking (Section 2.1), Android's web content integration mechanisms (Section 2.2), and the HyTrack attack (Section 2.3). Then, we introduce the concept of capabilities and capability-based security (Section 2.4), which forms the basis of our approach (Section 2.5). Finally, we summarize the threat model we assume throughout this thesis (Section 2.6).

2.1 Tracking on the Web and Mobile

Tracking mechanisms are typically divided into two broad categories: stateful and stateless tracking. Stateless tracking, also known as fingerprinting, infers a user's identity based on a combination of device-specific attributes [5]. Consequently, this method is hard to detect and block, but is also inherently less reliable, as small system changes may alter the fingerprint and disrupt identification.

Instead, stateful tracking relies on storing unique identifiers on the client device, most commonly through cookies or local storage. When a user revisits a site or interacts with embedded third-party content across domains, these identifiers are sent along with requests, allowing persistent recognition. While straightforward and highly effective, stateful tracking has become increasingly restricted through browser policies (e.g., third-party cookie blocking) and mobile platform changes such as the ability to disable the Google Advertising ID (GAID) on Android [6].

This problem not only affects the web, but also extends into the mobile ecosystem, as recently demonstrated by the Facebook Localhost Scandal [7] that exposed a covert tracking method used by Meta and Yandex on Android. In this case, their apps (e.g.,

Instagram) silently listened on localhost ports to receive browser tracking data, such as mobile browsing sessions and web cookies, sent from websites embedding Meta Pixel or Yandex scripts. This allowed the apps to link web activity to logged-in users, bypassing the browser's and Android's privacy protections. Although the practice was discontinued shortly after public disclosure, it highlighted a critical privacy gap between web content and native apps on mobile platforms.

2.2 Custom Tabs and Trusted Web Activities on Android

Capability	Browser	WebView	Custom Tab	Trusted Web Activity
Integration possible	○	●	●	●
Shares browser state	●[1]	○	●	●
Can hide URL bar	○	○	○	●
Can control size	○	●	◐[2]	○
Can open any URL	●	●	●	○[3]

○: unsupported, ●: supported, ◐: partially supported, 1: Owns the shared state, 2: Custom Tabs can be reduced to 50% programmatically and minimized by users, 3: Requires associatation via asset links.

Table 2.1: Overview of Web Content Integration Mechanisms on Android (Taken from Wessel et al.'s HyTrack paper [1]).

To integrate web content into Android applications, developers can use several mechanisms that differ in terms of security, performance, and user experience (Table 2.1). Among these, Custom Tabs (CTs) [2] and Trusted Web Activities (TWAs) [3] have emerged as the most popular alternatives to traditional WebViews [8], offering better performance and tighter integration with the user's default browser.

CTs were introduced to allow apps to display web content within the app's interface while leveraging the full capabilities of the user's browser. Unlike a WebView, which runs a separate, minimal web engine within the app, a CT is rendered by the installed browser itself. This means that all browser features such as optimized rendering, password managers, saved credentials, and cookies remain available. Developers can also customize the browser's UI elements, such as toolbar color and menu items, to visually align the CT with their app's theme. As a result, users perceive a seamless transition between native and web content without leaving the app context.

TWAs extend this concept further by removing nearly all browser UI elements, including the URL bar, and displaying web content in full-screen mode. This allows developers to integrate entire Progressive Web Apps (PWAs) or other web-based experiences into their native apps while maintaining a consistent appearance. For security reasons, launching a

TWA requires a Digital Asset Link (DAL) [9] — a mutual verification between the app and the website —, ensuring that both belong to the same trusted party. If this trust relationship cannot be verified, Android automatically downgrades the TWA to a regular CT.

A key advantage of both CTs and TWAs is that they share the browser's state. This means users can stay logged in to websites, reuse stored cookies, and maintain personalized settings across different apps and browsing sessions. This behavior improves usability and supports features like Single Sign-On (SSO), as authentication tokens from the browser can be reused within an app's embedded web view. However, as later discussed in Section 2.3, this same feature also introduces significant privacy risks. The shared cookie storage allows any app to access browser state information used by others, thereby enabling persistent cross-app and cross-web tracking techniques such as HyTrack.

In summary, CTs and TWAs offer a powerful bridge between the native and web ecosystems on Android. They combine the convenience and functionality of a full browser with the visual coherence of an app-embedded experience. Yet, the same integration that improves usability also blurs traditional security and privacy boundaries between apps and the web, which is exploited by for persistent-cross app tracking in the form of the HyTrack attack.

2.3 HyTrack Attack Overview

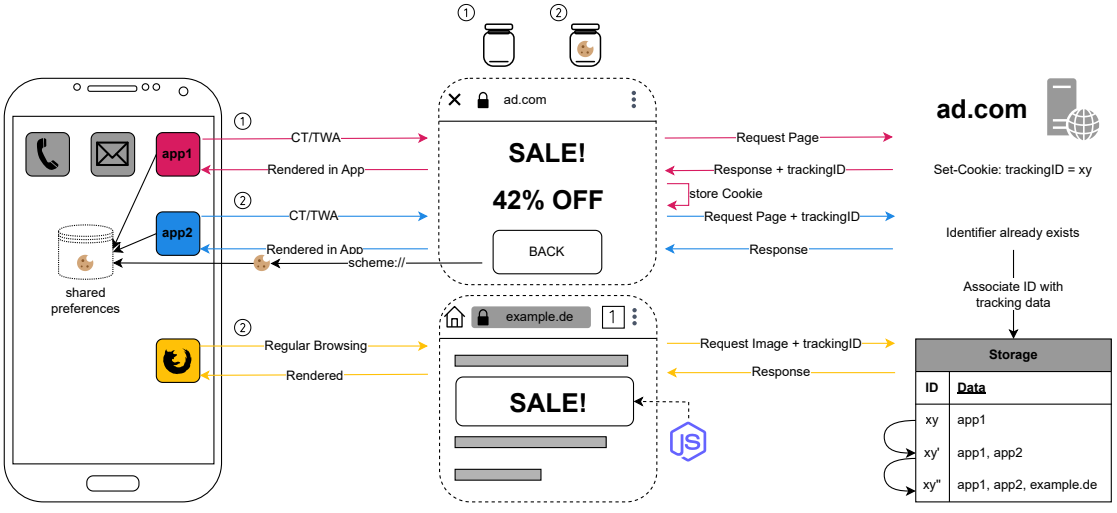


Figure 2.1: High-level Overview of the HyTrack Attack Flow (Adapted from Wessel et al.'s HyTrack paper [1]).

HyTrack, introduced by Wessels et al. [1], exposes a fundamental privacy flaw in this shared-state model. It demonstrates that third-party libraries embedded in multiple

apps can exploit the browser's global cookie storage to identify and track users across applications and even into their normal web browsing sessions. By leveraging standard Android features, rather than any explicit vulnerability, HyTrack highlights how the very mechanisms designed to make CTs and TWAs seamless for users can also undermine Android's app isolation guarantees.

Whenever an app opens a CT or TWA to display web content, the request is executed within the context of the user's default browser. This means that all cookies set by the visited domain are stored in the browser's global cookie jar and automatically reused in subsequent sessions, even if they originate from different apps. While this shared state enables seamless Single Sign-On and personalization, it also allows a tracking entity to correlate activity across multiple apps that interact with the same web domain.

HyTrack leverages this design as follows (Figure 2.1): a seemingly benign third-party library, included in several independent apps, opens a CT or TWA to a tracking domain controlled by the library's author. When this web page is first loaded, the server sets a unique identifier in a cookie, which is then stored in the shared browser state. When another app using the same library later opens a CT or TWA to the same tracking domain, the browser automatically attaches the existing cookie, thereby revealing that both apps are used by the same user. This creates a powerful cross-app identity link that persists outside Android's app sandbox and is invisible to both users.

Even more concerning, HyTrack's tracking identifiers are resilient to deletion, because the tracking library can store the identifier in the app's local storage and even use Google Play Services' backup feature to restore it after a device reset. In effect, HyTrack achieves Evercookie-like persistence at the system level, reviving deleted identifiers upon device restoration.

2.3.1 Weaknesses in Custom Tabs and Trusted Web Activities

The feasibility of this attack stems from three fundamental weaknesses in the current CT and TWA model:

1. **Implicit and Persistent Cookie Sharing:** All apps using CTs or TWAs share a single, persistent global browser cookie jar, regardless of developer intent. This shared state persists across app launches and user attempts to clear tracking data.
2. **Lack of App Context in the Browser:** The browser has no knowledge of which app initiated a given request and therefore cannot enforce app-specific cookie isolation or policy controls.

3. **Unrestricted Third-Party Inclusion:** Any third-party library that is embedded in multiple apps can launch CTs or TWAs, thereby accessing the shared browser state and enabling cross-app tracking.

By exploiting these design characteristics, HyTrack bridges the isolation between native and web contexts, effectively transforming legitimate web-integration features into a cross-app tracking channel.

2.3.2 Goals

The authors of HyTrack identified three essential goals that any practical mitigation against cross-app tracking must fulfill [1]. We adopt these goals as guiding principles for the design of our capability-based framework:

1. **Support for Web Platform Features:** Any mitigation should preserve the full functionality of web content, including support for cookies, JavaScript, and modern browser APIs.
2. **Seamless Integration:** The mitigation must operate transparently, without requiring additional user permissions or altering the normal app and browser experience.
3. **Controlled Access to Shared Browser State:** Isolation between applications must prevent tracking via shared state, while still allowing legitimate sharing scenarios such as Single Sign-On (SSO).

2.3.3 Mitigation Approaches proposed by HyTrack

The HyTrack authors outline two potential mitigation strategies and discuss the trade-offs each entails with respect to their design goals.

Browser State Partitioning. Browser state partitioning would allow each app to use its own cookie storage and hence prevent cross-app tracking. The seamless integration of web content remains intact, as no changes to the UI are necessary, but by completely removing the browser's shared state, benign uses like Single Sign-On (SSO) or ad personalization would be broken.

Forced User Interaction. In contrast, Forced User Interaction avoids this problem by explicitly requiring user consent before launching a CT or TWA. But this introduces a significant usability issue, as the user is forced to interact with the browser every time a web content is loaded, which not only degrades user experience but also breaks seamless integration of web content into the app. Furthermore, this approach hands control and responsibility to the user, which is not ideal from a security perspective, as the user might be unaware of the consequences of their actions and may inadvertently enable tracking by failing to interact with the browser as required.

Other strategies, such as limiting CTs and TWAs to First-Party Domains or disabling them for specific domains via browser options ultimately reflect the aforementioned approaches, relying on either browser state partitioning or forced user interaction. Therefore, these are not effective countermeasures against HyTrack.

2.4 Capabilities and Capability-based Security

A capability is an unforgeable and tamperproof token of authority that grants its holder specific access rights to a protected object. Unlike traditional access-control lists (ACLs), which base decisions on user or process identity, capabilities combine an object reference with an associated permission set, thereby enabling object-centric and decentralized access control. A capability system enforces the principle of least privilege (PoLP) by ensuring that possession of a capability is both necessary and sufficient for performing an operation on the referenced object. This follows the classic understanding established in early capability systems such as Hydra [10] and EROS [11], where capabilities are described as “prima facie evidence of authority”.

Because the right to access is embodied in the capability itself, no central authority needs to be consulted at the moment of use. Classical capability systems even allow controlled delegation by transferring capabilities between processes. In contrast, Byetrack deliberately prohibits capability transfer altogether, ensuring that capabilities remain bound to the originating application. Different implementations vary in how capabilities are stored, propagated, and revoked, yet all enable finer-grained control over authority than identity-based approaches.

2.5 Our Approach

In the context of this thesis, capability-based control offers an elegant way to isolate shared browser state and restrict the propagation of cookies, ensuring that only entities

explicitly possessing a valid capability may access a particular storage domain. The weaknesses are addressed as follows:

- **Explicit Cookie Isolation:** Cookies are stored only if a capability for the corresponding domain exists. Based on the access rights granted by the capability, cookies are either stored in the shared jar or returned to the app for isolated local storage.
- **App-Aware Browser Context:** Each capability encodes the app's identity and version, enabling the browser to enforce per-app cookie policies and invalidate outdated tokens.
- **Capability-Scoped Access Control:** Third-party domains without valid capabilities cannot access shared state, thereby blocking cross-app tracking. Legitimate use cases such as Single Sign-On (SSO) remain supported if the app developer explicitly allows them in the policy.

2.6 Threat Model

The HyTrack attack consists of three main parties: the app developer, the tracking company providing a third-party library, and the user. The tracking company aims to track the user across multiply apps and the web. For this, it provides a library that the app developer can include in their app, advertising it as an analytics or advertising SDK. The developer of an Android application unknowingly includes this library, which under the hood employs the HyTrack technique to conduct the tracking.

We want to prevent this library from tracking the app's user across multiple apps and empower app developer to use any third-party library without risking user privacy regarding cross-app tracking via HyTrack.

For this, we consider the app developer as benign but privacy-unaware. The app itself is untrusted after installation, as it includes the malicious tracking library, which can include arbitrary code with the same privileges. We assume the installer and the browser are trusted, as they initialize and enforce the mitigation.

Chapter 3

Methodology

The main research goal is to prevent HyTrack's cross-app tracking attack [1] by addressing the underlying weaknesses of CustomTabs (CTs) and TrustedWebActivities (TWAs) (Subsection 2.3.1), while adhering to the design goals defined by Wessels et al. (Subsection 2.3.2). Unlike mitigation strategies such as Browser State Partitioning or Forced User Interaction (Subsection 2.3.3), the proposed approach aims to maintain usability and compatibility while providing strong privacy guarantees.

The core idea is to encapsulate cookie access within fine-grained capability tokens, created and validated by the browser according to a developer-defined policy. Depending on the policy configuration, the app receives capabilities that either allow the domain's cookies to be shared across apps or restrict them to app-specific storage. When the app launches a URL in the browser, it transmits these capabilities along with the request. These capabilities determine how the browser handles cookies for that session, effectively isolating or sharing them based on the developer's intentions. This ensures that cookies from "trusted" domains intended for benign cross-app sharing are accessible to multiple apps, while cookies from "untrusted" domains remain isolated within each app's private storage and are only sent to the corresponding web server.

Consequently, this design prevents unauthorized cross-app tracking while preserving legitimate use cases such as Single Sign-On and session continuity. Because the system operates solely at the app–browser boundary, it maintains full compatibility with existing web platform features and requires no changes to the browser UI.

3.1 Capability Tokens

Because capabilities are delegable by design, we must bind them to the app's identity. Without this binding, a malicious third-party library embedded in multiple apps could mount a collusion attack by reusing capabilities issued to another more privileged app, thereby bypassing the mitigation entirely. Therefore, we include the app's unique package name in the capability tokens, so that the browser can verify that the token is only used by the app it was issued to. A global jar flag indicates whether the cookie belongs to the shared global jar or to the app-specific isolated jar. The cookie data (name and value) itself is encapsulated in the token, along with rights defining the permitted actions on the data later on (reading, writing, or both). Similar to JWTs [12], our tokens are signed to identify tampering attempts. Due to the threat model assumptions (Section 2.6), we additionally need to encrypt the tokens, so that a malicious third-party library embedded in the app cannot read the token contents and echo them back to its web server to circumvent the mitigation.

We distinguish between *Wildcard* and *Final* capability tokens. Final capabilities are fully specified tokens containing explicit cookie names and values. They stem from wildcard tokens and represent concrete cookie instances that are used directly for cookie enforcement. The wildcard tokens can be divided into *Classic Wildcard* and *Predefined Capabilities*.

The classic wildcard capabilities are partially specified tokens that omit cookie names and values. Predefined capabilities are wildcard tokens that specify cookie names but omit values, providing more granular control over individual cookies. Essentially, the wildcard tokens serve as templates from which final tokens are derived once cookies are received from a web server.

In conclusion, we design our capability tokens to be identity-bound and cryptographically protected data structures that encapsulate cookie information and metadata, enabling fine-grained access control between Android apps and the browser.

3.2 Capability-Based Cookie Isolation Flow

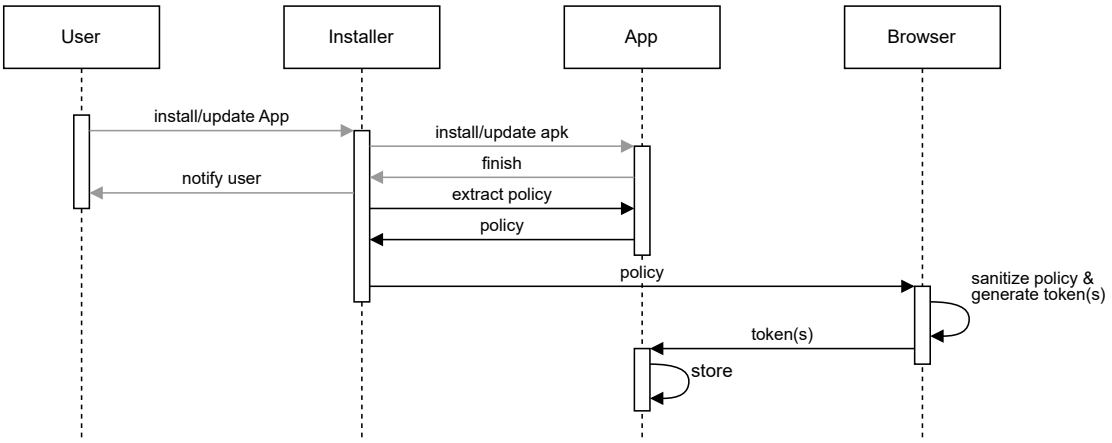
To achieve this, a prototype framework named Byetrack was designed, implemented, and evaluated on Android. The framework consists of three main components: a custom installer for policy extraction, a helper library for easier integration into the original AndroidX Browser library [13], and modifications to Firefox for Android (Fenix Project) [14] and its underlying GeckoView engine [15] to enforce cookie isolation via capability tokens.

To make our design unspoofable, we also needed to conduct modifications to the default Android operating system to securely bind capabilities to app identities.

3.2.1 Developer Policy

Developers define a JSON policy that specifies which domains may share browser state and, optionally, which cookies are expected from each domain. This allows granular control beyond simple trusted/untrusted domain distinctions, for example, isolating third-party cookies while permitting integration with a developer's own authentication domain or SSO provider. If no policy is provided, the browser falls back to "ambient mode", where all cookies are stored in the browser's shared cookie jar for backwards compatibility.

3.2.2 Capability Initialization



Note: The grey arrows represent the original installation flow.

Figure 3.1: Flow of capability initialization during app installation.

Figure 3.1 illustrates the flow of capability initialization during app installation. First, the installer extracts and transmits the policy to the browser. The browser validates and sanitizes the policy to ensure minimal privilege, removing conflicting or ambiguous entries. From the sanitized policy, the browser generates capability tokens as follows:

- For predefined cookie entries, the browser creates corresponding predefined capability tokens.
- For domain-level entries, the browser issues wildcard capabilities, marking them as global or private based on the policy.

- If no policy is provided, a single ambient capability is issued, reverting to the default shared-cookie behavior.

Each token is signed and encrypted before being sent to the app, which stores wildcard and final tokens in private storage for later use. When the app is updated, the installer retransmits the policy so that the browser can reissue capabilities consistent with the new app version.

Because the app must already be installed before its policy can be transmitted to the browser, there is a small window in which a race condition may occur: if the app launches a CT or TWA before the capability tokens have been issued and stored, the browser will receive no tokens and will therefore discard any cookies set during that initial request.

3.2.3 App–Browser Interaction

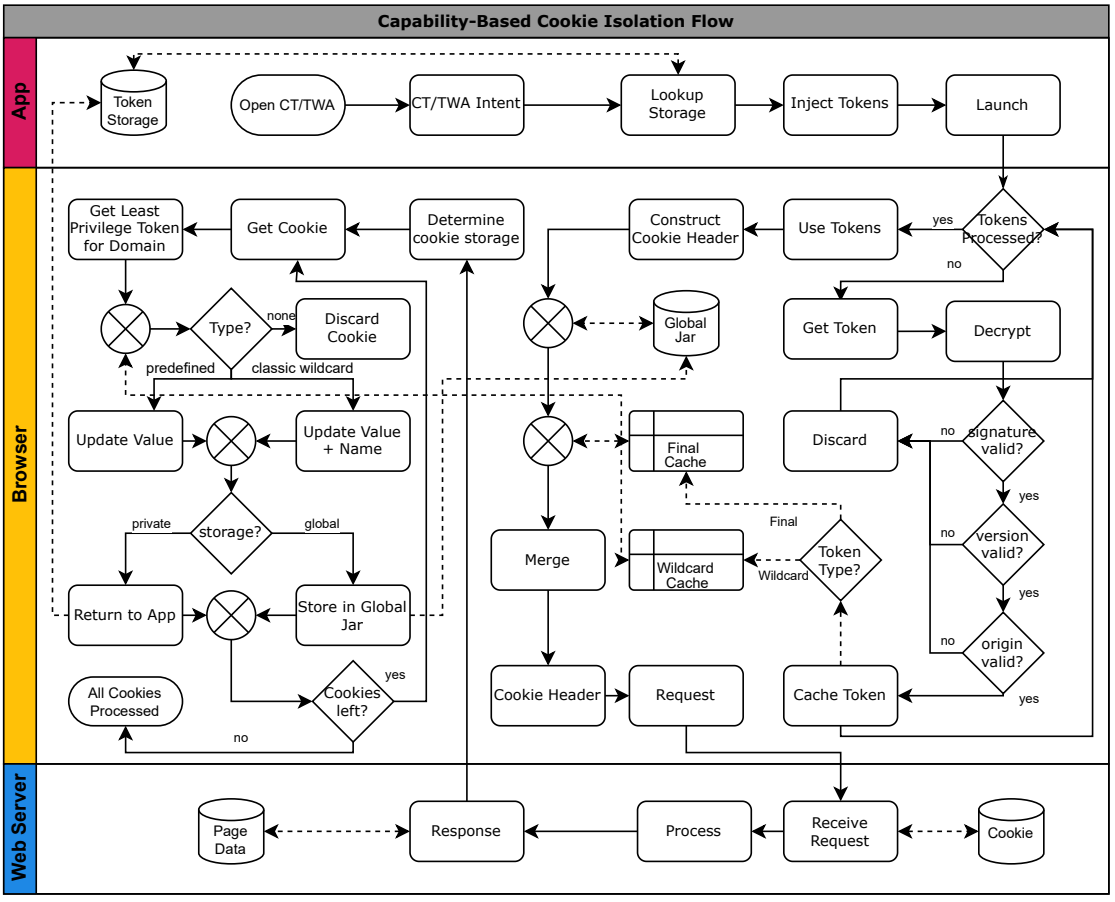


Figure 3.2: High-level overview of flow between app, browser and installer.

We describe the methodology flow of the app–browser interaction in our Byetrack framework for CT and TWA launches, as shown in Figure 3.2.

When an app opens a URL through a CT or TWA, the stored wildcard and (initially empty) final tokens are attached to the intent that launches the browser. Upon receipt, the browser decrypts and validates each token by checking its signature, package name and version number. Invalid tokens are discarded. The browser then uses the valid capabilities to determine how to store cookies received from the web server and to construct cookie headers for outgoing requests.

Cookie Reception. For every received cookie, the browser applies the following logic (in order of priority):

1. If the token is ambient, the cookie is stored in the global jar (default behavior).
2. If a private predefined capability matches the cookie name, the cookie value is filled in and returned to the app for local storage.
3. If a private wildcard capability exists, the cookie is filled in accordingly and returned to the app.
4. If a global predefined capability matches, the cookie is stored in the shared jar.
5. If a global wildcard capability exists, any cookie from the corresponding domain is stored in the shared jar.
6. If no capability matches, the cookie is discarded.

Cookie Transmission. When constructing requests, the browser merges the cookie header derived from the final capabilities with the cookies stored in the global jar. The latter appear in the global jar only when the app possesses a capability that authorizes global cookie storage, ensuring that each request reflects both app-specific and permitted shared state.

3.2.4 Utility Interfaces

To improve transparency and developer control, the browser exposes a small set of utility interfaces that allow an app to list the cookie names contained in its final capabilities and to read or update their corresponding values. These utilities are necessary, because without them, an app could not access the cookies stored in its isolated jar, as they are encapsulated in encrypted capability tokens that the app cannot interpret on its own.

Access to these interfaces is strictly governed by capability rights: reading a value requires read permission, while updating it requires write permission.

3.3 Design Advantages

Beyond preventing cross-app tracking, our design provides the following key benefits, while requiring only that developers adopt our modified browser library and include a simple JSON policy file in their app.

- **Fine-Grained Control:** Developers can precisely specify which cookies are shared or isolated.
- **Stateless Browser Design:** The browser remains stateless with respect to app-specific data, as apps retain and transmit their own tokens. The browser only needs to hold on to them temporarily during a session.
- **No Web Server Changes:** Web servers operate unmodified. The browser transparently enforces the capability model according to the app's policy it initially received.
- **Backwards Compatibility:** Apps without a policy fall back to the standard shared cookie behavior, ensuring compatibility with existing systems. Also, unmodified browsers continue to function correctly with apps using the framework, as unrecognized capabilities are ignored.

3.4 Alternative Design Considerations

An alternative architecture would delegate capability generation to the installer rather than the browser. This would simplify the browser's responsibilities to enforcement only, reducing its complexity and eliminating installer–browser communication for each app.

It would also solve the bootstrapping problem of generating the initial capabilities the current design faces: if an app is installed on a device, the installer notifies the browser of the new app and its policy, so the browser can generate capabilities accordingly. If this browser is not installed yet, the app cannot receive capabilities until the browser is installed and the app is reinstalled or updated.

A practical workaround for avoiding the need to reinstall or update existing apps after the browser's installation is to slightly adjust how browser applications are handled by our custom installer. A browser does not require any capabilities for cookie access, since it acts as the policy enforcement point itself. Consequently, when a browser application is installed, the installer can iterate over all already installed apps, collect their policies,

and forward them to the newly installed browser. The browser can then generate and issue the appropriate capabilities to each app retroactively.

Having the installer generate the capabilities directly would eliminate the need for this workaround, as it would allow an app to receive its capabilities immediately upon installation — independent of whether a browser is present or whether it supports the framework at all. However, this would require the installer and the browser to share a long-term secret key, introducing difficult key-distribution and key-management challenges and tightly coupling the security of both components. For this and simplicity reasons, the proof-of-concept implementation designates the browser as the sole trusted component for capability generation and enforcement and assumes the browser is present before app installation.

Chapter 4

Implementation

This chapter describes the implementation of our proof-of-concept prototype and details the individual components that together realize the Byetrack mitigation.

4.1 Policy Format

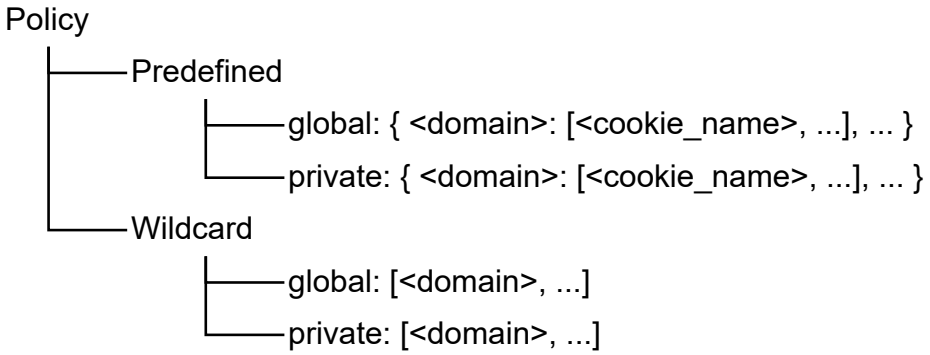


Figure 4.1: Policy file schema defining capability bindings for domains and cookies.

The Byetrack policy defines the configuration of capability tokens, specifying which domains may receive them and under which isolation context (global or private). It is expressed as a structured JSON object divided into two top-level sections: predefined and wildcard (Figure 4.1).

Each section further distinguishes between two isolation scopes:

1. **Global:** Cookies associated with these capabilities are stored in the browser's default cookie jar and are therefore shared across all apps using CTs/TWAs.
2. **Private:** Cookies associated with these capabilities are kept in the app-local cookie jar and are therefore not shared with other apps.

The predefined section specifies explicit capability bindings between domains and the cookies that are allowed to be associated with them. These entries define exactly which cookie names are permitted for which domains. Each key corresponds to a domain, and the associated list defines cookie names that are explicitly authorized for that domain.

The wildcard section defines simplified or implicit rules for domains where explicit cookie level definitions are not necessary. Instead of listing cookie names, the wildcard policy only specifies domains that shall receive capability tokens defined by the isolation scope.

The wildcard and predefined entries operate independently, thus a domain can appear in both lists if necessary. For example, a domain may have a global predefined token for a specific cookie and a private wildcard token for general use. The two scopes are treated independently and can coexist safely. Furthermore, the distinction between global and private in the predefined section allows a domain to hold both a global token and a private token. This allows flexible, layered control over cookie behavior. An example policy with explanation can be found in the appendix Section .1.

4.2 Capability Token Structure

Each capability token encodes metadata defining the scope and permissions associated with a cookie. The token is represented as a JSON object [16], as Firefox already provides helper utility for JSON parsing and serialization in the C++ layer to minimize implementation effort. Furthermore, JSON is a widely adopted format in web development, making it familiar to developers and easy to integrate with existing systems.

The object stores the following fields:

- `cookie_name`, `cookie_value`: The name and value of the cookie represented by this token.
- `domain`: The associated web domain this capability applies to.
- `application_id`: The package name of the application that owns the token.
- `app_version`: The version name of the application, used for version consistency checks.
- `rights`: The encoded access permissions (read, write, or none) granted for the cookie value.
- `global_jar`: Boolean flag indicating whether the cookie belongs to the shared or private cookie jar.

4.3 Custom Installer

Next to performing standard application installations, our custom installer is responsible for extracting each app's policy file and delivering it to the browser for token generation. To simplify deployment, the apps to be installed are bundled with the installer itself as APK files stored under its `assets/` directory. Alternative locations such as the `res/` folder are unsuitable, as resources are compiled into a binary format that cannot be directly accessed at runtime. Hosting the APKs remotely and downloading them on demand would also have been possible, but this would introduce unnecessary network dependencies and complicate reproducibility for our proof-of-concept.

Because the `assets` directory is read-only at runtime, each APK must first be copied into the installer's private file storage before it can be installed. The installation is then triggered by constructing an explicit Intent [17] that references the local file's Uri [18], sets its MIME type to `application/vnd.android.package-archive` [19], and grants temporary read permissions via `Intent.FLAG_GRANT_READ_URI_PERMISSION`. This intent is launched through a custom `ActivityResultLauncher` [20], which encapsulates Android's modern `ActivityResultContracts.StartActivityForResult` [21] mechanism.

This construct allows the installer to be notified directly when the installation flow finishes, without requiring any form of polling or background monitoring. Upon receiving a successful result code, the installer immediately proceeds to locate and read the policy file contained within the newly installed app's assets directory, using Android's `AssetManager` [22]. The extracted JSON policy is parsed and forwarded to the browser, together with the app's package name and version information obtained via the `PackageManager` [23]. For this to work, the package names of the apps to be installed must be registered in the Android Manifest [24] under the `queries` element [25].

For inter-process communication, we employ a `ContentProvider` [26] exposed by the browser. This IPC mechanism was chosen because it offers a straightforward implementation, automatically conveys the caller's UID, enabling reliable authentication of the requesting app and supports structured data transfer via Bundle objects [27].

Although Android provides functionality to determine the sender of a broadcast intent since API level 34 [28], we found these to be unreliable in practice, returning null for dynamically sent intents in our case. Using a `PendingIntent` [29] as a workaround would still be susceptible to spoofing, since a malicious app could craft and dispatch its own fake pending intent. A bound service [30] could also have served as a communication channel but would require the browser process to be running at the time of transmission and would considerably increase implementation complexity. Thus, the content provider

offered the most reliable and maintainable IPC solution for pushing the application's policy file to the browser.

4.4 Browser (Fenix)

We selected Mozilla Firefox for Android (Fenix) [14] as the browser base due to its open architecture and direct access to the GeckoView [15] engine. Its modifications fall into two main categories: (1) token generation and transmitting in `GeckoView`, Mozilla's WebView like Java library for Android that exposes the `Gecko` engine's functionality, and (2) the actual cookie enforcement within the C++ back-end of `Gecko`, the engine that powers the web browser [31, 32].

This separation maintains a clear trust boundary between high-level policy logic and low-level enforcement. Although their responsibilities overlap conceptually, especially regarding cryptographic operations on the tokens, direct invocation across these layers proved impractical, despite the possibility for `GeckoView` code to communicate between Java and C++ via the Java Native Interface (JNI) [33] (Subsection 4.7.2).

4.4.1 Token Generation

Global	Private	Allowed
predefined	wildcard	✓
wildcard	wildcard	× (take private)
wildcard	predefined	✓
predefined	predefined	× (take private)[1]

[1] *Note: Conflict if domain and cookie name match.*

Table 4.1: Allowed combinations of global and private policy entries.

Before generating any capability tokens, the browser performs a policy downgrade step to sanitize the received policy (Table 4.1). This step ensures that conflicting or overlapping entries are resolved according to the principle of least privilege (PoLP): private entries always take precedence over their global counterpart.

When the content provider receives a policy from the installer, it first parses the JSON structure into four collections directly corresponding to the four sections of the policy: predefined global, predefined private, wildcard global, and wildcard private. Each collection represents either a mapping from domains to explicit cookie names (for predefined entries), or a list of domains (for wildcard entries).

Predefined Conflict Detection. The first downgrade check handles domain-level and cookie-level conflicts within predefined entries. If a domain appears in both predefined global and predefined private for the same cookie name, the global cookie is removed, keeping only the private one. This logic is realized by iterating over the global map and comparing it to the private map and similarly for cookie-level checks.

Wildcard Conflict Detection. A similar check is applied to wildcard entries. If the same domain appears in both wildcard global and wildcard private, the global entry is discarded.

Independence Between Predefined and Wildcard Sections. Importantly, predefined and wildcard rules are treated independently. The downgrade logic explicitly avoids removing entries across these two categories. This means that a domain can safely appear in both sections with different privilege levels. This independence is reflected in the implementation by simply skipping cross-type downgrades. An example can be found in Section .1.

Once all conflicts are resolved, the downgraded policy structures are passed into the token generation routines – processing of the predefined map and wildcard list. These functions iterate over the filtered domain and cookie lists, creating one encrypted capability token per entry using `generateSingleToken(domain, cookie_name, "*", global_jar, package_name, version_name, rights)`. As a result, only conflict-free and least-privilege token objects of the format described in Section 4.2 are generated.

Note that the classic wildcard tokens omit the `cookie_name` and `cookie_value` fields, using "`*`" as placeholder value to indicate that they apply to all cookies for the given domain. Ambient tokens extend this paradigm further by also omitting the `domain` field, effectively applying to all domains. For wildcard tokens, the rights filed is set to `NONE` by default. This is due to the reason that wildcard tokens stay the way they are and are used as a blueprint for the browser to generate final tokens when cookies are actually received from the network and hence we do not want developers to accidentally overwrite the cookie value and thereby break the system.

Each token object is then encoded as a compact JSON object and serialized into a Base64-encoded string. Finally, the encoded string is signed using HMAC-SHA256 and the signature attached to the token object separated with a dot, similar to JWTs [12], before encrypting it using AES-CBC with a random IV using the browsers secret key. The signature makes the tokens tamper-evident, while encryption ensures confidentiality of the token contents.

Once generated, tokens are sent to the app in a map from domain to the String representation of the JSON array via the same content provider that received the policy so that the app can persist them locally (Subsection 4.5.1).

4.4.2 Launching Custom Tabs & TWAs with Tokens

The browser performs this process in two stages: (1) threading capability data from the Android layer down into the Gecko engine, and (2) enforcing cookie isolation inside Gecko's networking stack.

4.4.2.1 Threading of Capability Data

The threading mechanism ensures that all capability-related data (tokens and caller information) are propagated consistently through the Android and GeckoView layers until they reach the browser engine.

In Fenix, all incoming intents that trigger a custom tab launch are handled by the `CustomTabsIntentProcessor` class, which resides in the Android Components library, Mozilla's reusable collection of browser building blocks. Since Fenix currently does not support TWAs, any incoming TWA intent is downgraded to a standard custom tab intent and hence handled by the same processor.

Analogous to the existing `getAdditionalHeaders()` function that is used to retrieve custom HTTP headers to CT or TWA requests, we introduce a dedicated function that collects and returns all Byetrack-specific context data. This function retrieves the final and wildcard capability tokens, the UID of the calling application, a boolean flag and returns them as a structured map. We need the flag to distinguish between normal website launches initiated by the user and launches initiated by an app via CTs or TWAs, as only the latter should enforce Byetrack logic. Otherwise, the normal browsing experience would be affected as no cookies would be stored for normal website visits as the browser would not receive any tokens, and hence drop all cookies.

This map is then threaded through several layers of the Android Components architecture. Starting from the initial intent processing, it follows the custom tab launch path until it reaches the `GeckoEngineSession` class. `GeckoEngineSession` acts as a bridge between Android Components and GeckoView, Mozilla's Android library that exposes the Gecko browser engine APIs.

Within `GeckoEngineSession`, the data are handed over to the `GeckoSession` loader, the central entry point responsible for initiating page loads in GeckoView. Here, similar

to other loaders that handle fields such as `headerFilter`, `additionalHeaders`, or other `flags`, we introduced a new loader to transmit the capability tokens and UID.

Inside this loader, the `PackageManager` is used to derive the calling app's package name and version name from the UID passed in the intent. All these values (tokens, package name, and version name) are then encapsulated in a `GeckoBundle` — a lightweight key-value store optimized for inter-process communication between the Java and C++ layers of GeckoView.

The bundle is stored in the `GeckoSession` and attached to the `LoadUri` dispatch message. When this message is processed, the loader extracts the previously stored Byetrack fields and forwards them to the browser's `fixupAndLoadURIString` function. The parameters of this function are registered in the `LoadURIOptions` dictionary which holds load arguments for `docshell` loads. The docshell load parameters are initialized in the `DocShellLoadState` structure and carries functionality to get and set various load options we use in the Gecko's `DocumentLoadListener`.

Gecko's `DocumentLoadListener` is responsible for managing the lifecycle of document loads and connecting network responses with them, and therefore the ideal place to give the threaded opaque data meaning by parsing them back into usable tokens. During the document loading process, the Byetrack integration hooks into the `DocumentLoadListener` to process and apply capability tokens associated with a given navigation. The first step is carried out by the function `ProcessTokenBlob`, which transforms an incoming serialized token blob into validated `ByetrackToken` objects. Each blob is first parsed into its individual encrypted token strings, which are then decrypted into their JSON form. These JSON representations are deserialized into structured token objects and subsequently validated against the current application identity, consisting of the package name and app version. Tokens that fail to meet these validation criteria are discarded, ensuring that only authentic and context-appropriate capability tokens are propagated further in the loading process.

We implement a `ApplyByetrackFromLoadStateToBrowserContext` function that transfers our tokens from the `DocShellLoadState` into the active top level browsing context, where they become accessible throughout the browsing sessions lifecycle.

The function first verifies that no tokens or cookie headers have already been attached to the context, preventing redundant state updates. It then retrieves both the final and wildcard token blobs, along with the domain and package metadata, and processes them through the same parsing and validation pipeline. The final tokens are converted into a consolidated cookie header string, which is attached to the top level browsing context to be used by the network stack during request creation. Wildcard tokens, on

the other hand, are stored directly in the context's internal token array, allowing them to be evaluated dynamically for future requests.

Through this mechanism, the top level browsing context becomes the authoritative holder of Byetrack state for each navigation. It provides the cookie and network layers with all validated tokens and associated headers needed to enforce domain-scoped tracking policies. This design ensures that token verification occurs early in the navigation lifecycle while such that they only have to be parsed and validated once per navigation. This establishes a separation between token management and enforcement, allowing the network layer to focus solely on isolating cookies based on the pre-validated tokens.

4.4.2.2 Enforcement of Cookie Isolation

The actual enforcement of cookie isolation based on the capability tokens occurs in `Gecko`'s networking component `Necko` [34, 35], especially in its `CookieService` and `HttpBaseChannel`. When a network request is initiated, the `HttpBaseChannel` uses the `CookieService` to prepare the cookie header for outgoing requests and process incoming Set-Cookie headers from server responses and storing the cookies in the browser storage.

Outgoing Cookies. The main flow for attaching cookies to outgoing requests occurs in `HttpBaseChannel`'s `AddCookiesToRequest()`. Here, the `CookieService`'s `GetCookieStringFromHttp` function is called to retrieve the appropriate cookies for the target domain. Similarly, we use the top-level browsing context to retrieve our prebuilt cookie header string based on the final capabilities associated during the document load phase. The outgoing request header is constructed by appending our capability-based cookie string to the existing Cookie header, separated by a semicolon.

Incoming Cookies. Incoming `Set-Cookie` headers are handled by the `SetCookieHeaders` method of `HttpBaseChannel`, which processes server responses and stores cookies in the browser's cookie jar. In the original flow, this method iteratively invokes `SetCookieStringFromHttp` from `CookieService` for each cookie string. At this point, we intercept the flow and, using the top-level browsing context, retrieve the relevant wildcard capability tokens (including the "`enforcement`" flag), which we pass as an additional parameter to the cookie-handling logic.

This interception point is also where CHIPS [36] is implemented, making integration straightforward: CHIPS processing happens first, yielding a cookie annotated with its storage partition. We then feed this annotated cookie into Byetrack's decision-making logic.

The extracted information is passed to `DecideCookieAction()`, which constitutes the core of our evaluation logic. Before evaluating any token, the function inspects the "`enforcement`" flag to determine whether the cookie originated from a Custom Tab launch or from a normal browser-initiated navigation. If enforcement is disabled, the function immediately returns a decision to store the cookie normally, preserving the browser's default behavior.

Otherwise, all relevant capability tokens for the cookie's domain are evaluated according to the following precedence rules:

1. **Predefined tokens** always take priority over wildcard tokens.
2. **Private tokens** override global ones within the same category, ensuring stricter privacy when applicable.
3. **Wildcard tokens** apply only when no predefined token matches and define the handling of all cookies for the corresponding domain.

The result is encapsulated in a `ByetrackCookieDecision` object, which specifies both the chosen action (store, capture, or reject) and the token that authorized it. Based on this decision, the integration proceeds as follows:

1. **StoreNormally:** If the cookie is permitted for global storage (via a predefined or wildcard global token), Byetrack allows the native Gecko insertion flow (`storage->AddCookie()`) to proceed. This preserves expected behavior for legitimate cases such as session cookies or user preferences.
2. **CapturePredefined:** For cookies explicitly declared as private in the policy, the cookie value is embedded into the corresponding token, converting it into a final token. The token's access rights are elevated to `READ_WRITE`, enabling app-side access through Byetrack's utility interface (Subsection 4.4.3). Instead of being stored in the global jar, this updated token is forwarded to `StageTokenForReturn()`, which serializes and queues it for return to the originating app.
3. **CaptureWildcard:** For wildcard rules designating a domain as private, both the cookie name and value are captured into the token. Unlike predefined captures, the token's access rights remain at `NONE` to prevent a malicious library from simply reading the value and exfiltrating it, which would defeat isolation.
4. **Reject:** If no token matches the cookie or if the policy explicitly forbids handling cookies for the domain, the cookie is dropped, preventing unauthorized cross-site tracking.

This design ensures that regular web functionality remains intact while Byetrack transparently enforces fine-grained, policy-based cookie isolation.

Returning Tokens to the App. Before the token is serialized again and handed back up to the java layer to be sent to the app, we need to make sure that the token does not already exist in the app's private jar. For this, we fetch the cookie header stored in the top level browsing context and check if the cookie encapsulated by the token is already present in the header. This is done by re-constructing the standard cookie representation "`token.name=token.value`" and checking if it is contained in the merged cookie header stored in the top level browsing context, used for outgoing requests. If the token is found there, we discard it as the app already stores it in its private jar and there is no need to return it again. Otherwise, the token payload is serialized back into its JSON representation and then Base64-encoded, signed and encrypted the same way as during generation, before it is added to a temporary map stored in the `HttpBaseChannel` object by using its internal channel. This map associates each domain with an array of token strings, allowing multiple tokens for different domains to be staged for return to the application.

To synchronize the browser's enforcement results with the application process, we extend `Gecko`'s networking stack with a dedicated emission helper implemented in `HttpBaseChannel`'s `EmitByetrackTokensToGeckoView()`. This function is invoked from `HttpChannelParent`'s `OnStopRequest()`, that is, at the exact moment when an HTTP request completes and all cookies have been processed by the `CookieService`. Note that at this point, our subsystem has already collected any filled in tokens into the temporary map stored in the channel object by the `StageTokenForReturn()` function.

The `EmitByetrackTokensToGeckoView()` helper serializes this in-memory map into a compact JSON structure and forwards it through `Gecko`'s observer service. Before emitting, the function checks a boolean flag and a unique batch identifier to prevent re-emitting the same batch of tokens multiple times for a single channel instance. Using Mozilla's JSON writer utility, the function iterates over the map entries, associating each domain with an array of token strings. The output of the JSON follows the same structure as the one used during token generation, ensuring consistency between the two processes and thereby simplifying parsing on the receiving end. The function uses the global `nsIObserverService` to broadcast a notification with the topic "byetrack-final-tokens". This effectively acts as an IPC bridge between the networking layer in C++ and JavaScript/Java. After emitting the tokens, the internal map is cleared to avoid redundant emissions.

By performing this emission inside `HttpChannelParent`'s `OnStopRequest()`, we guarantee that the final set of captured tokens is only emitted once the HTTP transaction has completed and all cookies have been processed. This ensures that no partial or intermediate state is sent to the embedding application.

On the embedding side, the `byetrack-final-tokens` observer event is handled within `GeckoViewNavigation`, the same place where the threading in `GeckoView` started. The observer's `observe` method processes the serialized JSON map and uses `GeckoView`'s event dispatch system to send a message of type `GeckoView:Byetrack:FinalTokens` carrying the tokens and the application identity (package name). This event is caught on the Java side inside the responsible `GeckoSession`, the same location where `LoadUri` and similar events are processed. When the `GeckoView:Byetrack:FinalTokens` event is received, the Java handler constructs a `ContentValues` [37] object containing the token JSON and writes it to the application's registered `ContentProvider` [26], allowing it to update its local capability store.

4.4.3 Additional Utility

All additional utility functions are implemented in a separate `ContentProvider` [26] exposed by the browser. In this `ContentProvider`, we leverage the parameter `method` of the `call` function [38] to distinguish between the different utility functions. This also implies that all results are returned as a Bundle object, which is the standard return type of the call function. The data for each function to work on is passed via remaining parameters of the function (`arg` and `extras`) if necessary.

GetTokenNames. If the method parameter is set to `get_token_cookie_names`, the function expects a list of capability tokens in JSON array format as the `arg` parameter. Each token is decoded from its encrypted form using the `Token.decodeEncrypted()` function, which reconstructs the underlying `TokenPayload`. If the decoded token's `applicationId` does not match the caller's package name, the request is rejected. Otherwise, the function adds the mapping between the token string and its associated cookie name to the resulting `Bundle`, which is then returned to the caller. This enables external applications (e.g., the Byetrack client library) to inspect which cookie names are encapsulated by their issued capability tokens, without disclosing data from other apps.

GetTokenValue. If the method parameter is set to `get_token_cookie_value`, the function expects a single encrypted capability token as the `arg` parameter. Similar to the previous case, the token is decoded and verified against the caller's package name.

Afterwards, the browser verifies the access rights encoded within the capability token. The provider returns the cookie value only if the payload's is readable (`canRead() is true`), in which case the value is included in the "`value`" field of the result `Bundle`. Otherwise, a permission error string is returned. This design enforces read isolation and ensures that only apps possessing valid read rights for a specific capability can query associated cookie values.

WriteTokenValue. If the method parameter is set to `write_token_cookie_value`, the provider allows controlled modification of the cookie value embedded in a capability token. Again, the encrypted token is decoded, verified against the caller's package name, and its permissions checked via `canWrite()`. If write access is permitted, the new cookie value is taken from the `extras` bundle under the key "`value`". Since all fields of the payload are immutable, a new `TokenPayload` instance is created internally with the updated value, re-signed using the browser's signing key, and re-encrypted into a new token string. The updated token and its target domain are then returned to the caller. This process ensures that all token modifications remain cryptographically verifiable and bound to the correct application context.

4.5 App-side Integration

On the application side, we introduced two main components to enable seamless integration of Byetrack into existing Android apps: (1) a standalone Byetrack helper library that encapsulates capability token management, and (2) a customized AndroidX Browser library that automatically injects tokens into all CT and TWA launches. Together, these components ensure that apps using standard AndroidX interfaces transparently benefit from Byetrack's protection without code modifications.

4.5.1 Byetrack Helper Library

The Byetrack library acts as the bridge between the embedding app and the browser. It manages the storage, retrieval, and injection of capability tokens and exposes a minimal, high-level API through the `ByetrackClient` class. Internally, it consists of several modular components that collectively handle token management, secure communication with the browser, and intent preparation.

Token Management. To facilitate secure and persistent token handling, the library exposes a `ContentProvider` (`TokenProvider`) through which the browser can deliver

tokens to the application. This provider only implements the `insert()` [39] method, as neither querying nor deletion is required. When the browser calls `insert()`, the library first verifies the calling package name to ensure that only legitimate browsers can write to the app's token store. Upon successful verification, the transmitted tokens, typically provided as a key-value map containing both final and wildcard tokens are persisted locally.

Tokens are stored in `SharedPreferences` [40] under separate namespaces — `final_token`, `wildcard_token`, and `is_ambient` — to distinguish between different token types. The additional "ambient" flag indicates whether the app is currently operating in "ambient mode", which is crucial for correctly interpreting tokens that otherwise share the same structure.

`SharedPreferences` were chosen for simplicity, persistence across restarts, and asynchronous write support, characteristics well suited for our lightweight storage requirements. The `TokenManager` class abstracts all access to this local storage, offering thread-safe read and write operations and providing convenience wrappers to fetch or update specific token sets.

Token Injection into Intents. Before launching a browser instance, the app must attach its capability tokens to the outgoing intent. This is handled by the `ByetrackClient` via its `attachTokens()` and `injectTokens()` methods. When called, these methods gather all relevant tokens from the store, serialize them into a compact JSON bundle, and append them to the intent extras. If additional domains are supplied, the function ensures that the respective tokens are also included.

This injection step is completely transparent to the embedding app. Developers can use the same convenient methods `launchUrl()` [41] and `launchTrustedWebActivity()` [42] for launching a custom tabs intent and trusted web activity intent as before; the library automatically enriches them with the necessary Byetrack metadata prior to launch.

Utility Components. Supporting utilities such as `Util` and `DebugHelp` provide helper functions for (1) calling the browser's exposed content provider for additional operations on tokens and (2) displaying debug information about the current token stored and which cookies they encapsulate.

```
/**
 * Convenience method to launch a Custom Tabs Activity.
 * @param context The source Context.
 * @param url The URL to load in the Custom Tab.
 */
public void launchUrl(@NonNull Context context, @NonNull Uri url) {
    // Byetrack Hook before actually launching the Custom Tab
    ByetrackClient.attachTokens(intent, context, url, null);

    intent.setData(url);
    ContextCompat.startActivity(context, intent, startAnimationBundle);
}
```

Figure 4.2: Token injection in Custom Tabs launch function.

4.5.2 AndroidX Browser Integration

To eliminate the need for developers to manually include the Byetrack library or modify their own app code, we integrated it directly into a fork of the `AndroidX Browser` library [13]. This ensures automatic token injection whenever an app uses CTs or TWAs.

Specifically, both `CustomTabsIntent`'s `launchUrl()` and `TrustedWebActivityIntent`'s `launchTrustedWebActivity()` were extended to call the `attachTokens()` method of the exposed client before invoking the activity. Figure 4.2 shows this concretely for the `launchUrl()` function; the TWA launch function follows the same pattern. This guarantees that every navigation initiated through these standard `AndroidX` interfaces automatically carries the appropriate capability tokens to the browser.

Additionally, we introduced overloaded versions of both launch functions that accept an optional list of additional hosts, allowing developers to specify related domains that should receive tokens as well.

Overall, these modifications make Byetrack entirely transparent to app developers: any app compiled against our customized `AndroidX Browser` version inherently benefits from Byetrack's mitigation without requiring code changes or awareness of the underlying token system.

4.6 Integration into Existing HyTrack Demo Applications

Both HyTrack demo applications, `CrossAppLauncher` and `CrossAppTrackerOne`, originally launched TWA) using Chrome's `android-browser-helper` library [43], which automatically establishes a TWA connection to Chrome for convenience. To make these

apps compatible with our mitigation system, we removed this dependency and integrated our modified `AndroidX Browser` library instead. The TWA connection is now established manually with our customized Fenix browser.

For the launcher variant, we additionally implemented a standalone `TwaLauncherActivity` that launches a TWA on startup, since Firefox currently does not provide a comparable helper library like Chrome does. Although Firefox does not natively support TWAs yet, no further changes to the applications were required: the TWA intents are transparently downgraded to standard CT intents within the browser.

4.7 Implementation Challenges

The implementation process presented several notable challenges that required balancing security, practicality, and compatibility with existing Android mechanisms. This section summarizes the most critical ones encountered during development.

4.7.1 Securely Identifying the Calling Application

The most crucial challenge was securely identifying the application that launched a CT. This information is essential to verify that the capability tokens presented to the browser genuinely belong to the invoking application and are not spoofed by another app. In contrast, TWAs are not affected by this problem, as by design, they require a session with the browser over a Binder channel that inherently conveys the caller's identity.

Although Android's `Intent` mechanism provides an IPC channel between applications, it does not include a built-in way to authenticate the sender of an intent. We therefore explored multiple approaches to securely determine the calling application's identity, many of which turned out to be insecure or impractical within our threat model (Section 2.6).

Plain Intent Extras. The simplest approach was to attach the application's package name as an extra to the intent, similar to how capability tokens are passed. However, this method is fundamentally insecure since the package name is merely a string that can easily be spoofed by a malicious app pretending to be another package. Consequently, the malicious app could launch a CT with tokens belonging to the legitimate app, thereby bypassing our protections.

Shared Secret and Challenge-Response. A more advanced approach was to establish a shared secret between the application and the browser (e.g., using the `KeyStore` [44]) and

perform a challenge–response protocol during CT launch. This approach fails under our threat model, since a tracking library included in the app has the same privileges as the app itself and can therefore access the shared secret and execute the challenge–response on its own. Consequently, this mechanism cannot prevent colluding apps or libraries from impersonating the legitimate caller.

Using a Medium that Provides Caller Identity. We also explored mechanisms that inherently expose the caller's identity to the browser, such as using a Binder-based communication channel. However, deriving the identity from such a medium and linking it securely to the CT launch intent proved difficult, as any identifier accessible to the app is also accessible to the tracking library, and can thus be spoofed.

Commitment Scheme. Inspired by cryptographic commitment schemes [45], we considered a two-phase protocol involving a *commit* and a *reveal* phase. In this design, the application would first commit the intended URL and capability tokens via a secure channel from which the browser can derive the caller's identity. When the CT is later launched, the intent itself remains empty, and the browser uses the previously committed data.

While this design provides strong authenticity guarantees — the commitment occurs over a secure channel — it introduces significant complexity. The app must perform two separate actions (commit and reveal) and any other app could trigger a launch using previously committed data, as there is no strong link between the two phases.

Pending Intent. A more practical alternative was to leverage Android's `PendingIntent` mechanism [29]. In this approach, the intent originally used to launch the CT is wrapped into a `PendingIntent` flagged as immutable, and capability tokens are attached as extras. The browser can then obtain the caller identity securely via `getCreatorPackage()` or `getCreatorUid()` and invoke the `PendingIntent` on behalf of the app. Even if another app gains access to this `PendingIntent`, it cannot alter its content or spoof the original app's identity. This method therefore provides a viable balance between security and usability.

TWA-like Launch Modifications. Another potential approach was to redesign the CT launch mechanism to more closely resemble the session-based launch flow used by TWAs. In a TWA launch, the application must first bind to the browser's `CustomTabsService` [46] and establish a verified session. This binding step inherently conveys the caller's identity to the browser because it relies on an authenticated Binder IPC

connection rather than opaque intent extras etc. Therefore, one possible solution would have been to require all CT launches to follow the same pattern: before opening a URL, the app would be forced to bind to the `CustomTabsService`, obtain a validated session, and pass that session token to the browser when launching the Custom Tab.

Compatibility Considerations. Across all of these alternatives, a common issue emerges: each requires modifying how applications launch Custom Tabs. Whether through a two-phase commit–reveal sequence, wrapping the launch intent in an immutable `PendingIntent`, or enforcing a `TWA`-style session-binding flow, all approaches deviate from the standard `AndroidX Browser` contract. As a result, any solution along these lines would break backwards compatibility with existing applications that rely on the conventional intent-based Custom Tab launch mechanism. This implies that legacy apps that would now use our Byetrack-enabled browser library could no longer open CTs with a standard, non Byetrack-aware browser, limiting usability and adoption. For this reason, although these designs provide interesting security properties, they were ultimately not pursued.

Custom Android SDK Extension for Secure Caller Identity Propagation. To maintain compatibility with existing apps, we ultimately opted for a solution that allows the app to launch CTs using the standard API while still securely conveying the caller's identity. While the `ActivityTaskManagerService` [47] internally tracks the calling UID for permission checks and task management, this information is not encoded in the Intent object itself and therefore becomes inaccessible to the receiving application. To address this, we extended the Android framework with a custom, SDK-level enhancement that embeds the caller's UID directly into the Intent object. We introduced a new field `mRealCallingUid` along with a dedicated getter and setter. To ensure the field survives all typical Intent operations, including activity launches, redirections, and cross-process transmissions, we extended the intents copy-constructor, such as the parcel serialization and deserialization methods to handle this new field appropriately.

The next step was to ensure that the field is populated with the correct value inside the system server. We extended `ActivityStarter.startActivityInner()` [48], which is invoked during activity launches after the ActivityManagerServer [49] has fully resolved the activity and determined the real calling UID. Here, the `realCallingUid` is injected into the intent immediately before the system hands control to the target application (e.g., the browser). Since the field is part of the intent's serialized state, the target browser process reliably receives it without modification or interference.

This approach provides a trustworthy, unspoofable channel for conveying the caller's identity to the receiving browser, enabling capability-based cookie isolation without breaking compatibility with existing app behavior. However, this solution requires applications to compile against our customized Android SDK, which may limit adoption in practice.

Ideally, Android would natively expose the verified caller identity within `Intent`, which would eliminate the need for a custom SDK while retaining the security properties demonstrated by our implementation.

Android Identity Sharing Mechanism. Another possible direction would be to rely on Android's built-in identity-sharing mechanism, which allows an activity to learn the identity of its caller when launching a new activity. This can be enabled by setting `ActivityOptions.setShareIdentityEnabled(boolean)` to true [50].

However, adopting this mechanism would require every app that launches a CT to explicitly enable identity sharing and for the browser to reject any launch request that does not set this flag. This would break compatibility with existing apps, similar to the other approaches discussed above. If Android were to enable this flag by default for all CT launches, no additional SDK modifications would be necessary, as the underlying functionality already exists.

4.7.2 Bridging Between Java and Native Layers

In our design, token generation (Subsection 4.4.1), returning tokens to the app (Section 4.4.2.2), and querying the browser for additional token-related metadata (Subsection 4.4.3) are executed entirely within the Java layer, whereas enforcement is implemented inside the C++ network stack. Both layers require identical utility functions for JSON token parsing, encryption and decryption, and signature generation. From a software engineering perspective, the ideal architecture would place these utilities exclusively in the C++ layer and expose them to Java via JNI. This would provide a single source of truth and eliminate code duplication.

GeckoView supports such cross-layer calls through its `@WrapForNative` annotation, allowing Java code to invoke native C++ functions. However, these calls require an active `GeckoRuntime` instance [51]. In our case, token generation and content-provider based utility calls occur before the browser process is fully initialized, meaning no runtime is yet available. Attempting to temporarily bootstrap a runtime solely for token processing proved unreliable and introduced substantial complexity.

For these reasons, we ultimately separated responsibilities: the Java layer handles token creation, management, and policy interpretation, while the C++ layer enforces capability constraints once the browser process and its runtime are active. To maintain secure coordination without requiring direct JNI calls, both layers share a cryptographic secret known only to the browser. This enables the Java layer to prepare signed and encrypted token objects that the C++ enforcement layer can later validate without depending on shared execution context.

Chapter 5

Evaluation

In this chapter, we present the evaluation of our mitigation framework against the HyTrack cross-app tracking attack. We first describe the experimental setup (Section 5.1), before detailing (Section 5.2) and interpreting the results that we observed (Section 5.3).

5.1 Experimental Setup

For our setup, we emulated a standard Medium Phone with our modified Android 15 SDK (VanillaIceCream) installed with the proof-of-concept HyTrack applications and an additional test app.

5.1.1 HyTrack Applications

We base our evaluation on the two original HyTrack proof-of-concept applications provided by the authors: `CrossAppTrackerOne` and `CrossAppLauncher`. Each app was prepared in two variants: (1) a baseline version without any policy, replicating the original HyTrack attack scenario, and (2) a mitigated version including a developer-defined policy that enforces cookie isolation for the tracking domain.

Integrating our mitigation framework into these apps required no code changes to the application logic. Developers only need to replace the standard AndroidX Browser dependency with our modified version that supports the capability-based mitigation mechanism and add a policy file. Instead of installing the apps directly, we used our custom installer, which extracts and forwards the policy to the browser for token generation and issuance.

Simply switching to our enhanced Firefox browser was insufficient, Because the original HyTrack apps rely on the Android Browser Helper library [43] for launching TWAs — a library that (among others) is tailored for establishing TWAs with Chrome for convenience. Therefore, we removed the Android Browser Helper dependency and established the required session connection to Firefox manually, allowing the apps to open TWAs (or CT fallbacks) in our modified Fenix browser.

5.1.2 Test Application

To gain deeper insight into the runtime behavior of our framework, we implemented an additional test application. This app displays all capability tokens received from the browser when launching a CT or TWA to a specific domain, along with the corresponding cookies they encapsulate.

The app provides dedicated controls to launch both private and global domains (defined in its policy) to observe the browser's cookie-handling behavior. It also illustrates policy downgrading for ambiguous configurations, and allows testing of reading and writing cookie values via the issued capabilities. This setup provides a controlled environment to validate the correct isolation and usage of capability tokens.

5.2 Results

Our results focus on three main aspects: the mitigation of the HyTrack attack, verification of the design goals postulated by the authors of HyTrack, and highlighting the additional behaviors our mitigation provides.

5.2.1 Mitigation of HyTrack

We first replicated the HyTrack attack under baseline conditions. When both HyTrack apps were installed without a policy, the tracking domain successfully set and retrieved a persistent identifier cookie shared across both applications, confirming the presence of cross-app tracking as described in the original work.

After applying our mitigation framework with a policy marking the tracking domain as private, this behavior was eliminated. The browser correctly issued a capability to the app encapsulating the tracking cookie, instead of storing it in its global jar. Consequently, when the second app opened a TWA to the same domain, no cookie was sent, and the

tracking domain issued a new identifier. Each app thus maintained an independent cookie context, effectively breaking the cross-app tracking channel.

Subsequent requests within each app still reused their local capability tokens, preserving session continuity within the same app. These results demonstrate that our mitigation successfully isolates cookie state across apps while preserving per-app continuity — a key goal of our approach.

5.2.2 Verification of Design Goals

Our system was designed with three primary goals in mind, as outlined in Chapter 3. The following evaluation confirms that our implementation satisfies these goals.

1. **Support for Web Platform Features:** All standard web platform features, including cookies, JavaScript, and modern APIs, remained fully functional throughout our evaluation, only the cookie storage behavior was altered.
2. **Seamless Integration:** The mitigation operates transparently without affecting the user interface of the app or browser. Launching TWAs and Custom Tabs required no additional steps, and transitions between native and web content remained smooth.
3. **Controlled Access to Shared Browser State:** Domains defined as private (predefined or wildcard) in the policy were correctly isolated from the global cookie jar. Trusted domains (e.g., SSO providers) remained accessible via the shared jar, allowing legitimate cross-app scenarios such as Single Sign-On to continue functioning as expected.

These observations confirm that our mitigation maintains compatibility with web features, integrates seamlessly, and provides reliable control over access to shared browser state.

5.2.3 Additional Behavioral Verification

Beyond mitigating HyTrack, our evaluation shows that developers can express fine-grained, domain- and cookie-specific isolation rules directly through the policy file. Cookies marked as private were correctly withheld from the global jar and instead returned to the app via encapsulated capabilities. When an app was installed without a policy, the browser fell back to issuing an ambient capability, thereby preserving full backward compatibility. Likewise, applications providing policies but running on an

unmodified browser continued to operate normally, as unsupported capabilities were safely ignored. As a side benefit, this separation also enables apps to perform web-based logins (via CT/TWA) using an account different from the one logged in within the browser itself. The integration effort for developers remained minimal: replacing the browser library dependency and adding a small policy file were sufficient to enable the mitigation.

Overall, these experiments demonstrate that the framework behaves consistently across various configurations and can be adopted with very low developer overhead.

5.3 Interpretation of Results

The evaluation results demonstrate that our capability-based mitigation framework effectively prevents cross-app tracking via HyTrack without sacrificing web functionality or developer usability. By enforcing cookie isolation through app-defined policies, our system successfully breaks the cross-app tracking channel while maintaining per-app continuity and legitimate browser behavior. This confirms that capability-based access control, when integrated at the application–browser boundary, is a practical and efficient method to achieve fine-grained privacy guarantees without redesigning core browser storage mechanisms.

Our findings further indicate that the use of a policy-driven model provides a clear balance between privacy and flexibility. Developers can selectively isolate untrusted domains while continuing to rely on the shared global cookie jar for legitimate cases such as SSO. This result suggests that privacy enforcement can be delegated to the developer, rather than statically enforced by the browser, while still ensuring strong isolation guarantees for users.

Chapter 6

Discussion

In this chapter, we discuss the broader implications of the results presented in the previous chapter, focusing on how our findings affect developer workflows (Section 6.1), compatibility with existing browser mechanisms (Section 6.2), usability considerations (Section 6.3), and performance aspects of the proposed framework (Section 6.4).

6.1 Developer Empowerment and Transparency

Beyond mitigating HyTrack's tracking capabilities, our approach introduces a new dimension of developer transparency and control. Developers can explicitly define which domains and cookies should remain private, preventing accidental leakage of sensitive identifiers to third parties. This transforms the browser from a monolithic storage manager into a configurable privacy mediator, empowering developers to reason about and enforce their privacy boundaries.

The ability to read and modify issued capability tokens directly from within the app also enables explicit auditing and debugging of cookie behavior, which may be particularly beneficial during development and testing. This developer-centric perspective distinguishes our approach from existing browser-level isolation mechanisms, which operate opaquely and offer little insight into how cookies are handled internally.

6.2 Compatibility with Existing Mechanisms

Our mitigation mechanism is designed to coexist with, and complement, existing browser-level cookie isolation techniques such as CHIPS [36]. If a capability authorizes access to the shared cookie jar, cookies are stored using Firefox's native partitioning logic.

Otherwise, storage occurs solely in the app's local context. This hybrid model preserves the benefits of CHIPS while adding a developer-driven control layer on top, allowing flexible yet secure isolation boundaries. Such compatibility is critical for deployability, as it allows gradual integration into browsers without requiring the removal or modification of existing standards.

6.3 Usability and Adoption Considerations

The usability and adoption of Byetrack depend largely on how the necessary platform changes are handled. If Android were to natively expose the caller UID in Intents — as demonstrated in our prototype implementation —, our mitigation introduces virtually no additional friction for developers or end-users. In this scenario, developers only need to include a policy file and depend on our modified AndroidX Browser library, as all remaining enforcement occurs transparently in the browser. No new APIs, permissions, or configuration steps are required.

This low integration effort enables incremental adoption: developers can choose to isolate only selected domains or even specific cookies without modifying their existing codebase. End-users benefit from stronger privacy guarantees without any visible change in app or browser behavior.

Under such platform support, the framework integrates naturally into the existing Android ecosystem. A future release of the AndroidX Browser library could embed this logic directly, allowing widespread deployment without imposing additional work on developers.

If, however, Android does not provide the caller UID in Intents by default, developers would need to build against a modified SDK to enable capability issuance. This requirement raises the barrier to adoption and may limit practical deployment. Nonetheless, even in this case, the developer-side integration remains minimal once the platform support is present.

A minor usability limitation concerns app updates: when a policy is retransmitted and new tokens are issued, cookies previously stored in an app's isolated jar are discarded.

6.4 Performance Overhead

Letting the browser merely sign the tokens before transmitting them to the application does not provide sufficient security (Section 2.6). Therefore, our design requires the

Scope	Wildcard	Predefined
Private	highest	low-high
Public	lowest	low

Table 6.1: Priority levels of Byetrack capability tokens based on their scope and definition type.

browser to both encrypt and decrypt capability tokens during issuance and validation. This approach ensures end-to-end integrity and confidentiality but inevitably introduces performance overhead due to the additional cryptographic operations. The magnitude of this overhead depends primarily on the number of cookies processed, the type of capability token employed, and the defined access scope (Table 6.1). By storing the tokens by domain in the app, we already minimize the number of tokens needed to be processed for both cookie reception and transmission.

Public wildcard. If a domain is classified as trusted, the browser issues a single public wildcard token covering all cookies associated with that domain. As a result, the performance overhead is minimal: only one token must be decrypted and verified, regardless of the number of cookies. All cookies share the same access rights to the global cookie jar and therefore follow the browser's native storage logic without additional isolation steps.

Private wildcard. If a domain is deemed untrusted, the browser issues one private wildcard token for all cookies originating from that domain. Again, only a single token must be decrypted for verification. However, unlike the public case, all cookies are stored in the app's isolated jar. This increases overhead as the number of cookies grows, since each cookie must be individually wrapped in a capability and subsequently signed and encrypted by the browser for local storage.

Public predefined. If individual cookies are explicitly registered as public in the policy, the browser generates separate predefined tokens for each cookie for access to the shared jar. This leads to higher overhead during issuance, as each cookie requires its own token to be signed and encrypted. During verification, each predefined token must also be decrypted and validated individually. Nevertheless, since these cookies remain in the shared jar, the recurring runtime cost for subsequent accesses is relatively low.

Private predefined. For domains in the policy declared as private, where individual cookies are defined explicitly, the browser issues one private predefined token per cookie.

This configuration incurs the highest overall cost: every token must be decrypted and validated separately, and each cookie must be re-encrypted and stored in the app's private jar. While providing the strongest isolation guarantees, this mode also introduces the greatest cryptographic overhead.

6.5 Limitations

While our mitigation addresses HyTrack's cross-app tracking channel effectively, several limitations remain. First, the system relies on correctly defined policies; incomplete or misconfigured policies may lead to under- or over-isolation, affecting usability or privacy respectively. Second, integrating the approach on the application side is straightforward by just replacing the browser library dependency, but extending the approach to other browsers would require their cooperation and modification, as browsers use different engines and therefore might have different cookie management mechanisms. Despite this, the changes would still be relatively small, as most modern browsers are Chromium-based and share similar architectures.

Moreover, our threat model assumes non-malicious developers. If an app developer intentionally collaborates with a tracking library or exfiltrates tokens, the current design cannot prevent data leakage. Similarly, by storing the capability tokens encrypted in the app's private storage, we can prevent a malicious third-party library from reading and modifying them easily, but we cannot prevent the library from deleting the tokens themselves, as both the app and the library are executed under the same UID and thus inherit the host app's privileges. Nonetheless, such deletion would only result in loss of session continuity rather than cross-app tracking.

6.6 Summary

Overall, the discussion highlights that capability-based, policy-driven cookie isolation offers a practical path to mitigating cross-app tracking while maintaining the flexibility required for legitimate web integrations. The results confirm that it is possible to reconcile privacy and usability within the app–browser ecosystem, providing developers with meaningful control over cookie behavior.

Chapter 7

Related Work

HyTrack [1] demonstrates a novel cross-app and cross-web tracking technique in the Android ecosystem by exploiting the shared cookie storage between CTs and TWAs. This allows persistent tracking of users across multiple applications and the browser, even surviving user efforts to reset or sanitize their environments.

The need to address HyTrack becomes even more critical in light of additional research. Beer et al. [52] conducted a comprehensive security analysis of CTs and revealed that they can be exploited for state inference, SameSite cookie bypass, and UI-based phishing attacks. Their work further shows that Custom Tabs are widely adopted, with over 83% of top Android apps using them, often via embedded libraries. These findings reinforce that CTs are a high-value attack surface and that the shared browser state — central to HyTrack — has broader security implications. As TWAs are a specialized form of CTs, they are similarly affected, further enabling the tracking to be fully disguised.

While HyTrack highlights a serious privacy vulnerability, no concrete mitigation has been proposed that balances privacy with the legitimate need for seamless web integration within mobile apps, such as SSO ad delivery.

Modern browsers prominently adopt state partitioning to combat third-party tracking. Firefox's Total Cookie Protection (TCP) [53] and Safari's Intelligent Tracking Prevention (ITP) [54] both enforce per-site cookie jars, thereby limiting cookie-based cross-site tracking. However, this also breaks legitimate third-party services that rely on shared cookies, such as SSO or ad personalization.

Google is actively working on a similar mechanism under the name CHIPS (Cookies Having Independent Partitioned State) [36]. CHIPS allows third-party cookies to be partitioned by the top-level site with an optional `Partitioned` flag, enabling legitimate services like SSO to maintain function while avoiding broad tracking vectors. However,

CHIPS is not applicable to Android's embedded web contents like CTs or TWAs, as the top-level site can be the tracker itself. Our solution can be seen as extending this paradigm to the app level.

Our interpretation of capability tokens is inspired by JSON Web Tokens (JWTs) [12], which are widely used in web authentication to encode claims about a user or a session in a secure, verifiable manner. Instead of storing user information directly on the server upon receiving a POST request, JWTs allow the server to issue a signed token that contains the necessary claims, which the client can then present in subsequent requests. As a result, the server does not need to maintain session state, as the token itself carries all the information needed and can verify via the signature that the token has not been tampered with. For this purpose, JWTs consist of three components separated by dots: a header that specifies the token type and algorithm for encoding and decoding it, a payload for the actual data, and a signature of the first two parts after base64 encoding that ensures the integrity of the token. We extend this idea by including the cookie information and other metadata in the token's payload, and by establishing a communication channel between the browser and the app: the browser issues these tokens according to the app's policy, and the app presents them in subsequent requests to either access the browser's shared cookie jar or store cookies in its own app-local storage.

The work of Georgiev et al. titled "Breaking and Fixing Origin-Based Access Control in Hybrid Web Applications" [55] highlights critical failures in how hybrid apps enforce origin boundaries. Specifically, they show that WebViews and hybrid frameworks often bypass or misapply the Same-Origin Policy (SOP), enabling attackers to inject or reuse authentication tokens across apps and domains. Their proposed mitigation involves reintroducing stricter origin enforcement tied to app identities. Our approach builds on this idea by using capability tokens to encode both the origin and the app context explicitly, thereby preventing unauthorized reuse or delegation.

Chapter 8

Future Work

This thesis has laid the groundwork for mitigating cross-app tracking through shared browser state on Android by introducing a capability-based access control framework. However, several directions for future research and development remain.

First, the proposed framework should be implemented and evaluated across additional browsers to assess its generalizability beyond the Mozilla ecosystem. In particular, reproducing the experiments for the browsers analyzed by Wessels et al. in their HyTrack study [1] (Opera, Firefox, Tor Browser, UC Browser, Brave, and others) would help to determine cross-browser compatibility and highlight potential implementation challenges in differing browser architectures.

Second, the applicability of the framework should be explored on other platforms and devices, such as iOS or desktop environments, where similar shared-state mechanisms may facilitate cross-context tracking. Extending capability-based cookie isolation to these ecosystems could provide a unified approach to mitigating state-based tracking across mobile and web platforms.

Third, future work may examine the usability and performance trade-offs of capability enforcement, developer adoption barriers, and possible integration with emerging web privacy standards such as CHIPS or Storage Partitioning. Such investigations would contribute to a more holistic understanding of how capability-based isolation can strengthen user privacy without impairing the functionality of embedded web technologies.

Finally, for simplicity, the current prototype preserves only a cookie's name and value. Other attributes, such as expiration time, Secure and HttpOnly flags, SameSite policies, and path or domain scope, are not yet encoded in capability tokens. Incorporating these attributes would enable full reproduction of cookie semantics and avoid subtle

mismatches between the browser's internal cookie model and the capabilities used for isolated storage.

Chapter 9

Conclusion

Android's Custom Tabs and Trusted Web Activities enable seamless integration of web content into Android applications while leveraging browser features such as cookies, storage, and Single Sign-On. Although this enhances user experience, the shared browser state introduces privacy risks, as demonstrated by Wessel et al. [1] through the HyTrack attack, which enables persistent cross-app tracking across Android apps and the web.

This thesis presents Byetrack, a capability-based framework designed to mitigate such cross-app tracking attacks by introducing policy-based, per-application control over shared browser state. Application developers can define fine-grained policies that specify which domains are trusted to access shared browser data. During installation, the framework automatically generates cryptographically bound capability tokens for each application, which the browser verifies and enforces at runtime. Through this mechanism, Byetrack isolates cookies and related identifiers according to developer-defined trust boundaries, effectively limiting cross-app information leakage without breaking legitimate functionality.

Byetrack extends the AndroidX Browser Library and Firefox for Android to integrate policy enforcement directly into the Custom Tabs and Trusted Web Activity workflows. Developers can further benefit by designating first-party cookies as private or by predefining specific trusted cookie names, allowing precise control over shared browser state. Experimental evaluation confirmed that the framework prevents unauthorized cookie sharing between apps while preserving compatibility with intended Single Sign-On scenarios.

In summary, this work demonstrates that capability-based access control provides a practical and effective foundation for mitigating cross-app tracking through shared browser state on Android. Byetrack shows that stronger privacy guarantees can coexist

with usability and interoperability, marking a step toward a more privacy-conscious mobile web ecosystem.

Bibliography

[1] M. Wessels, S. Koch, J. Drescher, L. Bettels, D. Klein, and M. Johns, "Hytrack: Resurrectable and persistent tracking across android apps and the web," in *34th USENIX Security Symposium (USENIX Security 25)*. Seattle, WA: USENIX Association, Aug. 2025.

[2] "Android developers custom tabs," https://developer.android.com/develop/ui/views/layout/webapps/overview-of-android-custom-tabs, accessed: 2025-11-16.

[3] "Android developers trusted web activities," https://developer.android.com/develop/ui/views/layout/webapps/trusted-web-activities, accessed: 2025-11-16.

[4] S. Kamkar, "Evercookie," *URl: http://samy. pl/evercookie*, 2010.

[5] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020.

[6] "Play console help google advertising id," https://support.google.com/googleplay/android-developer/answer/6048248?hl=en, accessed: 2025-11-17.

[7] LocalLeaks, "Tracking users with localhost: Facebook's covert redirect abuse," https://localmess.github.io/, 2023.

[8] "Android api reference webview," https://developer.android.com/reference/android/webkit/WebView, accessed: 2025-11-16.

[9] "Android developers digital asset links," https://developer.android.com/training/app-links/verify-applinks#auto-verification, accessed: 2025-11-16.

[10] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Communications of the ACM*, vol. 17, no. 6, pp. 337–345, 1974.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: a fast capability system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999, pp. 170–185.

[12] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7519

[13] "Android api reference androidx.browser," https://developer.android.com/jetpack/androidx/releases/browser, accessed: 2025-11-12.

[14] "Firefox source docs firefox for android (fenix)," https://firefox-source-docs.mozilla.org/mobile/android/geckoview/contributor/geckoview-architecture.html#front-end-and-back-end, accessed: 2025-11-16.

[15] "Firefox source docs geckoview," https://firefox-source-docs.mozilla.org/mobile/android/geckoview/index.html#geckoview, accessed: 2025-11-16.

[16] "Android api reference jsonobject," https://developer.android.com/reference/org/json/JSONObject, accessed: 2025-11-09.

[17] "Android api reference intent," https://developer.android.com/reference/android/content/Intent, accessed: 2025-11-10.

[18] "Android api reference uri," https://developer.android.com/reference/android/net/Uri, accessed: 2025-11-10.

[19] "Android api reference intent.setdataandtype," https://developer.android.com/reference/android/content/Intent#setDataAndType(android.net.Uri,%20java.lang.String), accessed: 2025-11-10.

[20] "Android api reference activityresultlauncher," https://developer.android.com/reference/androidx/activity/result/ActivityResultLauncher, accessed: 2025-11-10.

[21] "Android api reference activityresultlauncher," https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts.StartActivityForResult, accessed: 2025-11-10.

[22] "Android api reference assetmanager," https://developer.android.com/reference/android/content/res/AssetManager, accessed: 2025-11-10.

[23] "Android api reference packagemanager," https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts.StartActivityForResult, accessed: 2025-11-10.

[24] "Android api developers manifest," https://developer.android.com/guide/topics/manifest/manifest-intro, accessed: 2025-11-18.

[25] "Android api reference manifest <queries>," https://developer.android.com/guide/topics/manifest/queries-element, accessed: 2025-11-10.

[26] "Android api reference contentprovider," https://developer.android.com/reference/android/content/ContentProvider, accessed: 2025-11-10.

[27] "Android api reference bundle," https://developer.android.com/reference/android/os/Bundle, accessed: 2025-11-10.

[28] "Android api reference broadcastreceiver.getsendfrompackage," https://developer.android.com/reference/android/content/BroadcastReceiver#getSentFromPackage(), accessed: 2025-11-10.

[29] "Android api reference pendingintents," https://developer.android.com/reference/android/app/PendingIntent, accessed: 2025-10-25.

[30] "Android api reference bound service," https://developer.android.com/develop/background-work/services/bound-services, accessed: 2025-11-10.

[31] "Firefox source docs front-end and back-end separation," https://firefox-source-docs.mozilla.org/mobile/android/geckoview/contributor/geckoview-architecture.html#front-end-and-back-end, accessed: 2025-11-11.

[32] "Firefox source docs gecko engine," https://firefox-source-docs.mozilla.org/overview/gecko.html#gecko, accessed: 2025-11-11.

[33] "Firefox source docs java native interface (jni)," https://firefox-source-docs.mozilla.org/mobile/android/geckoview/contributor/geckoview-architecture.html#java-native-interface-jni, accessed: 2025-11-11.

[34] "Mozilla wiki networking," https://wiki.mozilla.org/Networking, accessed: 2025-11-11.

[35] "Firefox source docs networking," https://firefox-source-docs.mozilla.org/networking/index.html#networking, accessed: 2025-11-11.

[36] Google, "Cookies having independent partitioned state (chips)," https://github.com/privacycg/CHIPS, 2023.

[37] "Android api reference contentvalues," https://developer.android.com/reference/android/content/ContentValues, accessed: 2025-11-12.

[38] "Android api reference contentprovider call method," https://developer.android.com/reference/android/content/ContentProvider#call(java.lang.String,%20java.lang.String,%20java.lang.String,%20android.os.Bundle), accessed: 2025-12-10.

[39] "Android api reference contentprovider insert method," https://developer.android.com/reference/android/content/ContentProvider#insert(android.net.Uri,%20android.content.ContentValues,%20android.os.Bundle), accessed: 2025-11-12.

[40] "Android api reference sharedpreferences," https://developer.android.com/reference/android/content/SharedPreferenceshttps://developer.android.com/reference/android/content/SharedPreferences, accessed: 2025-11-12.

[41] "Android api reference customtabsintent launchurl method," https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsIntent#launchUrl(android.content.Context,android.net.Uri), accessed: 2025-11-12.

[42] "Android api reference trustedwebactivityintent launchtrustedwebactivity method," https://developer.android.com/reference/androidx/browser/trusted/TrustedWebActivityIntent#launchTrustedWebActivity(android.content.Context), accessed: 2025-11-12.

[43] "Chrome android browser helper library," https://developer.chrome.com/docs/android/trusted-web-activity/android-browser-helper-migration, accessed: 2025-11-13.

[44] "Android api developers keystore," https://developer.android.com/privacy-and-security/keystore, accessed: 2025-11-17.

[45] Wikipedia contributors, "Commitment scheme — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Commitment_scheme&oldid=1318450859, 2025, [Online; accessed 25-October-2025].

[46] "Android api reference customtabsservice," https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsService, accessed: 2025-11-13.

[47] "Android google open source activitytaskmanagerservice," https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ActivityTaskManager.java, accessed: 2025-11-13.

[48] "Android google open source activitytaskmanagerservice," https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/wm/ActivityStarter.java, accessed: 2025-11-13.

[49] "Android api reference activitymanagerservice," https://developer.android.com/reference/android/app/ActivityManager, accessed: 2025-11-13.

[50] "Android api reference activityoptions setshareidentityenabled," https://developer.android.com/reference/android/app/ActivityOptions#setShareIdentityEnabled(boolean), accessed: 2025-11-17.

[51] "Firefox source docs jni runtime delegates," https://firefox-source-docs.mozilla.org/mobile/android/geckoview/contributor/geckoview-architecture.html#calling-runtime-delegates-from-native-code, accessed: 2025-11-13.

[52] P. Beer, M. Squarcina, L. Veronese, and M. Lindorfer, "Tabbed out: Subverting the android custom tab security model," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 4591–4609.

[53] Mozilla, "Firefox's total cookie protection," 2021, https://developer.mozilla.org/en-US/docs/Web/Privacy/State_Partitioning.

[54] Apple, "Intelligent tracking prevention," 2020, https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.

[55] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *NDSS symposium*, vol. 2014, 2014, p. 1.

Appendix

.1 Example Policy File

```
{
  "predefined": {
    "global": {
      "royaleapi.com": ["__royaleapi_session_v2",
    ↪ "another_cookie"]
    },
    "private": {
      "schnellnochraviolimachen.de": ["named_cookie"],
      "royaleapi.com": ["__royaleapi_session_v2"]
    }
  },
  "wildcard": {
    "global": [
      "royaleapi.com"
    ],
    "private": [
      "nr-data.net"
    ]
  }
}
```

- *royaleapi.com* only receives a predefined private and a global wildcard token (for general cookie usage). This is because the identical cookie *__royaleapi_session_v2* of the same domain is registered to receive a token for both isolation scopes. The token generator therefore downgrades the token to the private one, as it is more restrictive.
- *schnellnochraviolimachen.de* receives a private predefined token limited to one cookie.

- *nr-data.net* receives a private wildcard token, granting limited cookie handling rights without predefined cookie names. In this scenario, *nr-data.net* is a third-party domain embedded in the first-party website *royaleapi.com*.

.2 Use of Generative Digital Assistants

During this thesis, I used several generative AI tools within the allowed scope (code generation, literature lookup, debugging help, and text revision).

Claude Sonnet 4.0, integrated into Visual Studio Code, supported me in navigating and understanding the Firefox/GeckoView codebase. It helped me locate relevant files, follow control flows, and identify functions involved in specific behaviors—tasks that would have been very time-consuming manually. Claude was also useful when writing the JSON token parser and serializer in Java and C++, and when selecting and using appropriate cryptographic libraries (e.g., HMAC-SHA256, AES-CBC with random IVs).

Both ChatGPT (GPT-4/5 models) and Claude were used for debugging support by explaining unclear compiler or runtime errors and suggesting fixes. While some suggestions (especially related to JNI) turned out to be unhelpful, others were surprisingly effective. Notably, ChatGPT helped point me toward ActivityStarter as the correct place in Android's source code to propagate the caller UID into an Intent.

GitHub Copilot was used for code completion and to speed up writing boilerplate code, common Android patterns, and repetitive structures.

For writing the thesis itself, I used ChatGPT to rephrase and improve technical sections for clarity, structure, and readability.

Overall, these tools helped speed up the development and writing process by reducing the time needed to search through large codebases, debug complex issues, and refine written text.