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

Bachelorthesis

Capabilities as a Solution against Tracking Across Android Apps

submitted by

Tim Christmann

on November 17, 2025

Reviewers

Dr. Sven Bugiel

Noah Mauthe

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, November 17, 2025,

(Tim Christmann)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, November 17, 2025,

(Tim Christmann)

Write dedication here

Abstract

Trusted Web Activities and Custom Tabs 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 shown by HyTrack, this integration also introduces severe privacy risks by blurring the boundary between web and app contexts, allowing persistent tracking through the browser's shared cookie storage.

In this work, we propose 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 servers are confined to isolated, in-app cookie jars. This empowers well-meaning developers to continue leveraging third-party libraries while preventing them from performing unauthorized cross-app tracking. At the same time, essential features such as Single Sign-On and personalized content delivery remain fully functional. Our approach balances privacy and usability, allowing tracking-resistant web-app integration without degrading the user experience.

Acknowledgements

I would deeply like to thank Dr. Sven Bugiel and Noah Mauthe for their support and guidance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, Android applications have increasingly leveraged web content within their interfaces to enhance user experience and streamline features such as authentication and monetization. To enable this, developers often use Custom Tabs (CTs) and Trusted Web Activities (TWAs), technologies that provide seamless, browser-backed web integration while maintaining native-like performance and features. This approach allows web-based functionality like Single Sign-On (SSO), such as login via Facebook or embedded advertising, without forcing users to switch between app and browser.

However, these benefits come at a cost. CTs and TWAs share the browser's cookie storage across all apps, enabling continuity of web sessions – but also opening serious privacy vulnerabilities. Recent research by Wessels et al. introduced HyTrack [1], a novel tracking technique that exploits this shared browser state to persistently track users across different applications and the web, even surviving device changes, cookie clearing, or browser switching. HyTrack works by embedding a third-party library into multiple unrelated apps. Each app, unaware of the library's true purpose, opens a CT or TWA to the same tracking domain. This domain sets a unique identifier in a cookie, stored in the browser's shared cookie jar. When another app using the same library loads content from the same domain, the cookie is sent, enabling the tracker to correlate activity across apps and even into regular browser use. Due to Android's backup mechanisms, the tracking ID can be restored even after a factory reset, rendering it more persistent than the evercookie [2].

This thesis explores whether capabilities, a fine-grained access control model, can be used to limit or prevent these privacy issues without breaking legitimate use cases of CTs and TWAs. Specifically, we aim to design and evaluate a framework that allows developers to retain the benefits of third-party libraries (e.g., SSO or monetization) without exposing users to invisible, cross-app tracking. The framework should be simple to integrate,

practical in real-world deployments, and minimize interference with already established app workflows.

Chapter 2

Background

2.1 Capabilities

What are Capabilities? how to classify our capabilities -> identity-based cryptographic

2.2 Custom Tabs and Trusted Web Activities on Android

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

Custom Tabs 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 Custom Tab 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 Custom Tab with their app's theme. As a result, users perceive a seamless transition between native and web content without leaving the app context.

Trusted Web Activities (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) – 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 Custom Tab.

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 – intentionally or not – to access browser state information used by others, thereby enabling persistent cross-app and cross-web tracking techniques such as HyTrack.

In summary, Custom Tabs and Trusted Web Activities 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

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: a seemingly benign third-party library, included in several independent apps, silently 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 and app developers.

Even more concerning, HyTrack's tracking identifiers are resilient to deletion. Because Android's automatic backup mechanisms restore application data, including browser-managed cookies, the tracking identifier can survive browser resets, app reinstallations, and even factory resets. In effect, HyTrack achieves evercookie-like persistence at the system level, reviving deleted identifiers upon device restoration.

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

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.

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.

Unrestricted Third-Party Inclusion: Any third-party library embedded across multiple apps can open CTs or TWAs, gaining access to the shared browser state and enabling tracking across unrelated apps.

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.

Chapter 3

System Design

To mitigate cross-app tracking via HyTrack, we introduce a developer-defined policy mechanism enforced by cryptographically secured capability tokens that govern cookie sharing and isolation on a per-app basis. This chapter describes the design of this capability-based approach, the structure of the capability tokens, their enforcement model, and the overall system flow between the installer, app, browser, and web servers.

3.1 Design Overview

Our system builds on the concept of capabilities – unforgeable tokens that grant their holder specific rights to access a resource. Unlike traditional access control lists (ACLs), capabilities enable a decentralized and fine-grained access model, as the right to perform an operation is encoded directly within the token rather than managed by a central authority. This means the browser can enforce cookie access based on the capabilities presented by the app, without maintaining a global permission mapping of app identities to cookies.

Byetrack applies this principle to browser state management: cookies are encapsulated within capability tokens that specify how and under which conditions they can be accessed or stored. The browser acts as a Policy Enforcement Point (PEP), while the app, guided by a developer-provided policy, receives only the capabilities necessary for its legitimate functionality.

3.2 Capability Token Structure

Each capability token encodes metadata defining the scope and permissions associated with a cookie. The following fields form the basis of our design:

- **Cookie Name and Value:** Contain the actual cookie data managed by the browser.
- **Signature:** Ensures the capability was issued by the browser and has not been modified.
- **Package Name:** Identifies the app that owns the capability. This prevents implicit delegation—without it, a receiving app could reuse capabilities issued to another app.
- **Domain:** Specifies the target web server. The browser enforces that cookies are only valid for this domain, preventing malicious libraries from reusing capabilities to store untrusted cookies in the global jar.
- **App Version Number:** Allows the browser to detect outdated tokens after app updates. Whenever the app is updated, the installer retransmits the policy so the browser can issue new capabilities consistent with the new version.
- **Rights:** Define the permitted actions—reading, writing, or both—on cookie data. These rights prevent tracking libraries from exploiting browser access to extract or misuse capability contents.
- **Global Jar Flag:** Indicates whether the cookie belongs to the shared global jar or to the app-specific isolated jar.

3.3 Capability Types

We distinguish between *final*, *wildcard*, and *ambient* capabilities.

- **Final Capabilities:** Fully specified tokens containing explicit cookie names and values. They represent concrete cookie instances and are used directly for cookie enforcement.
- **Wildcard Capabilities:** Partially specified tokens that omit cookie names or values. They serve as templates from which final tokens are derived once cookies are received from a web server.

- **Ambient Capabilities:** Represent the browser's default behavior—storing all cookies in the shared global jar. These act as fallbacks when no explicit policy is provided and offer no privacy guarantees.

3.4 Security and Enforcement Design

Byetrack makes browser state access explicit, app-aware, and scoped through the following principles:

- **Explicit Cookie Isolation:** Cookies are stored only if a capability for the corresponding domain exists. Based on the capability's global flag, 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 by granting appropriate global capabilities.

3.5 The Threat Model

The developer of an Android application unknowingly includes a third-party library that uses the HyTrack technique for their own purposes, such as advertising. We want to prevent this library from tracking the apps user across multiple apps and empower the app developer to use any third-party library without risking user privacy in regards to cross-app tracking via HyTrack.

For this, we assume that the app developer is not malicious and does not intend to violate user privacy. Otherwise, developers could simply choose to omit using our mitigation framework and directly use the HyTrack library on their will.

A trusted component is the installer. Next to installing the app, it also extracts the app's policy and hands it of to the (trusted) browser, the Polcy Enforcement Point (PEP). The browser initially generates the capability tokens according to the app's policy and sends them to the app, which stores them in private storage.

As the tracking library is included in the app, it has the same permissions as the app itsef, which means it can include arbitrary code, for example attempt to modify tokens or policies. Additionally, we have to assume collaboration between the tracking library and other apps to share stored tokens and meta data of the mitigation framework. Attemps such as sending policy to their own benefit and thus circumventing the mitigation are also possible.

As we hook our defense in the androidx browser library, any developer that wants to use the malicous tracking library – or any other library that relies on Custom Tabs or Trusted Web Activities – automatically uses our mitigatin framework. Thus, the developer cannot choose to omit the mitigation, but still disable it by not giving a policy at all. Therefore, only the androidx browser library needs to be updated, instead of relying on the developer to additionally include the mitigation library, which could be forgotten or omitted intentionally.

3.6 Methodology

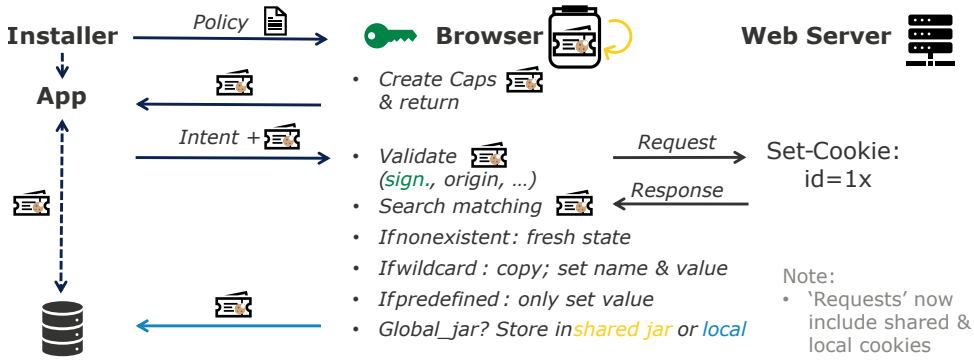


Figure 3.1: High-level overview of the Byetrack flow between installer, app, browser, and web servers.

3.6.1 Developer Policy Flow

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.

If no policy is provided, the browser falls back to ambient mode, where all cookies are stored in the shared jar for backwards compatibility.

3.6.2 Capability Initialization

During app installation, the installer extracts and transmits the policy to the browser. Before issuing any capability tokens, 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.

3.6.3 App–Browser Interaction

When an app opens a URL through a Custom Tab (CT) or Trusted Web Activity (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, version number, and target domain. Invalid tokens are discarded.

The browser then uses the valid capabilities to:

1. Determine how to store cookies received from the web server.
2. 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 cookies derived from the app's valid final tokens with those from its global jar, ensuring that each request accurately reflects both app-specific and shared state according to the developer policy.

3.6.4 Utility Interfaces

To improve transparency and developer control, the browser exposes limited utility functions that allow the app to:

1) Retrieve the names of cookies encapsulated in final capabilities.
2) Read their corresponding values.
3) Write or update cookie values.

Access to these utilities is strictly controlled through capability rights: read operations require read rights, and modifications require write rights.

3.7 Design Advantages

Beyond preventing cross-app tracking, Byetrack offers several key benefits:

B1) **Fine-Grained Control:** Developers can precisely specify which cookies are shared or isolated.

B2) **Stateless Browser Design:** The browser remains stateless with respect to app-specific data, as apps retain and transmit their own tokens.

B3) **No Web Server Changes:** Web servers operate unmodified—the browser transparently enforces the capability model.

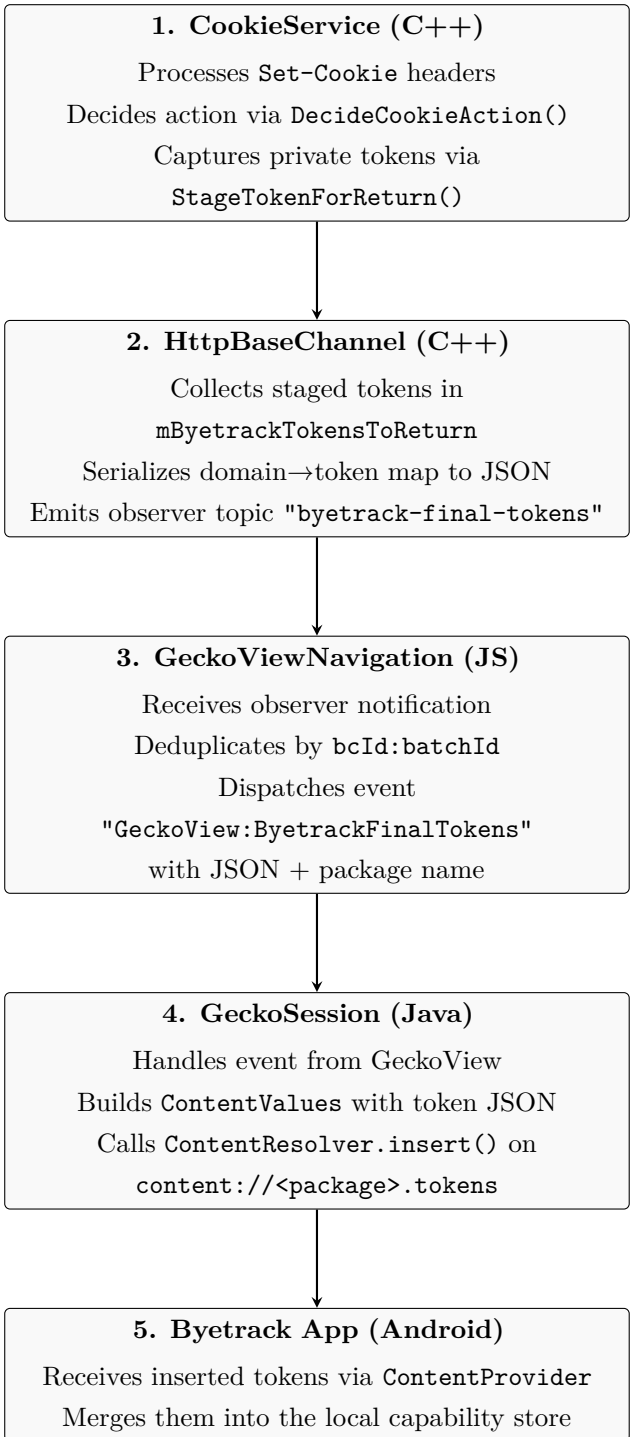
B4) **Backwards Compatibility:** Apps without a policy fall back to the standard shared cookie behavior, ensuring compatibility with existing systems.

3.8 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. However, it would require shared cryptographic secrets between the installer and browser, thereby enlarging the trusted computing base and attack surface. For these reasons, the proof-of-concept implementation designates the browser as the sole trusted component for capability generation and enforcement.

Chapter 4

Implementation



4.1 Overview

This chapter details the implementation of our proof-of-concept prototype that realizes the Byetrack mitigation described in the previous chapter. The prototype consists of three main components that together cover policy distribution, capability token generation, and enforcement within the browser:

- Custom Installer Application — responsible for installing test apps and transmitting their declared policies to the browser.
- Byetrack Helper Library — a reusable client-side library that manages capability tokens, communication with the browser, and transparent token injection into outgoing intents.
- Modified AndroidX Browser Library — a drop-in replacement for the standard AndroidX Browser module that automatically integrates Byetrack logic into every *Custom Tab (CT)* and *Trusted Web Activity (TWA)* invocation.

For the enforcement component, we selected Mozilla Firefox for Android (Fenix) as the browser base due to its open architecture and direct access to the GeckoView engine. Within Fenix, Byetrack is implemented across two layers: a Java layer that handles policy ingestion and token generation, and a native C++ layer within Gecko responsible for enforcing cookie isolation and network-level restrictions. Both layers communicate through structured JNI bindings and share a common cryptographic key for token signing and verification.

To demonstrate interoperability, we integrated the system into the original HyTrack demo applications (`CrossAppLauncher` and `CrossAppTrackerOne`) and developed an additional test app as well as an adversarial tracking app. This allowed us to evaluate the behavior of our enforcement logic across benign, mixed, and malicious interaction scenarios.

4.2 Policy Format

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*.

Each section further distinguishes between two isolation scopes:

- global – referring to tokens or capabilities that are valid across all browser profiles or trusted applications (e.g. legitimate SSO domains).
- private – referring to tokens restricted to the local application or site context (e.g. third-party trackers like the authors of HyTrack describe).

Predefined Section: The predefined section specifies explicit capability bindings between domains ad 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 distinction between global and private in the predefined section allows a domain to hold both a global token and a private token. The two scopes are treated independently and can coexist safely.

Wildcard Section: 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 – 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. This allows flexible, layered control over cookie behavior.

An example policy with explanaition can be found in the appendix 5.

4.3 Custom Installer

The installer application packages the proof-of-concept apps as APKs within its `assets/` directory. Since this directory is read-only at runtime, the APKs are first copied into the app's private storage before installation. Each installation is initiated through an intent pointing to the local APK URI.

Once installation completes, the installer immediately proceeds to read the policy file from the newly installed app's assets using the `AssetManager`. To avoid polling for completion, the process is implemented via `ActivityForResult`, allowing direct continuation after installation finishes.

After parsing the policy JSON into a string, the installer transmits it to the browser—along with the package name and version obtained via the `PackageManager`—through a designated `ContentProvider` exposed by the browser. We selected a content provider for

inter-process communication because it is simple to implement, automatically conveys the caller's identity, and provides structured data exchange via Bundles.

Although recent Android versions introduce mechanisms to derive the calling package of a broadcast intent, we found this approach unreliable in practice (often returning `null`). Wrapping the intent in a `PendingIntent` is a possible workaround but susceptible to spoofing if a malicious library creates and dispatches its own fake pending intent. A bound service could also serve as an IPC channel but would require the browser process to be active and significantly increase implementation complexity.

4.4 Browser (Fenix)

The browser serves as the policy enforcement point in our architecture. Its modifications fall into two distinct components: (1) a Java layer responsible for token generation and management, and (2) a native C++ layer responsible for enforcement within Gecko's cookie and networking subsystems.

This separation maintains a clear trust boundary between high-level policy logic and low-level enforcement. Although their responsibilities overlap conceptually, direct invocation across these layers proved impractical, as the browser may receive policies before its `GeckoRuntime` is initialized. Launching a temporary runtime solely for token processing introduced instability and race conditions. Therefore, the two layers share a symmetric secret key to ensure coordinated behavior without direct cross-layer calls.

4.4.1 Token Generation

Before generating any capability tokens, the browser performs a policy downgrade step to sanitize the received policy. This step is implemented inside the TokenGenerator.generateCapabilityTokens() function and ensures that conflicting or overlapping entries are resolved according to a "minimal security" principle — meaning that private entries always take precedence over global ones, and predefined and wildcard rules remain independent.

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, the global entry is removed entirely. If the same cookie name is found under both sections for the same domain, 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 the appendix 5.

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, cookieName, "*", globalJar, packageName, versionName, rights). As a result, only conflict-free and least-privilege token objects of the following format are generated:

- `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 issuing 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.
- `signature`: HMAC-SHA256 signature over all fields, ensuring integrity and authenticity.

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 hmacSHA256 and the signature attached to the token object separated with a dot, similar to JWTs [3], before encrypting it using AES-CBC with a random IV using the browsers secret key. This makes the tokens tamper-evident and ensures that only the browser can generate valid tokens.

Once generated, tokens are send to the app in a map from String (domain) to JSON array via the same content provider that received the policy so that the app can persist them locally (Section **??**).

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.

Stage 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 Firefox for Android (Fenix), all incoming intents that trigger a Custom Tab (CT) 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 Trusted Web Activities (TWAs), any incoming TWA intent is downgraded to a standard Custom Tab intent and handled by the same processor.

Analogous to the existing getAdditionalHeaders() function – used to retrieve and attach custom HTTP headers to CT or TWA requests – we introduced 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 flag "true"

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 Custom Tabs 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 (where the Custom Tabs logic resides) 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 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 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 Document-LoadListener to process and apply capability tokens associated with a given navigation. The first step is carried out by the ProcessTokenBlob function, 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, version, and target domain. 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 from the DocShellLoadState into the active top level top level browsing context, where they become accessible thoughout the top level browsing 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.

Stage 2 – Enforcement of Cookie Isolation. The actual enforcement of cookie isolation based on the capability tokens occurs in the CookieService and HttpBaseChannel class.

When a network request is initiated, the HttpBaseChannel uses the CookieService to (1) prepare the Cookie header for outgoing requests and (2) 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::AddCookiesToRequest`. Here, the `CookieService::GetCookieStringFromHttp` function retrieves 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. For incoming `Set-Cookie` headers, `HttpBaseChannel::SetCookieHeaders` processes and stores cookies received from server responses. It calls `CookieService::SetCookieStringFromHttp` iteratively for each cookie string and stores them in the browser's cookie jar. Here, we again leverage the top-level browsing context to retrieve wildcard capability tokens such as the "enforcement" flag and pass them as an additional parameter. This location also hosts the CHIPS [4] implementation, making it straightforward to integrate our logic: CHIPS is evaluated first, and only if it allows storage does the Byetrack logic apply next.

When a response containing cookies is received, our implementation first extracts the cookie name and value from the nsCookie object and logs this information alongside the associated base domain and the number of active tokens currently available for that site. These tokens represent the capability-based permissions granted to the application according to its installed policy, such as whether specific domains or cookies may be stored globally or privately.

The extracted information is then passed to the function DecideCookieAction(), which encapsulates the core of our decision-making logic. Before any evaluation occurs, the function checks the "enforcement" flag whether Byetrack enforcement is enabled for this session. If enforcement is disabled, the function immediately returns a decision to store the cookie normally, preserving default browser behavior. Otherwise, all available tokens for the current cookie are evaluated based on the following precedence rules:

1. Predefined tokens have absolute priority over wildcard tokens.
2. Within the same class, private tokens override global ones, ensuring that stricter privacy rules are always applied when present.
3. Wildcard tokens apply when no predefined token is available and indicate that all cookies from that domain should be handled according to their declared scope (global or private).

The result is a ByetrackCookieDecision object that specifies both the action to take (e.g., store, capture, or reject) and the corresponding token that granted the decision.

Depending on this decision, the integration proceeds as follows:

1. StoreNormally: If the cookie is authorized for global storage (e.g., by a predefined or wildcard global token), Byetrack simply continues the native Gecko cookie insertion flow via the storage->AddCookie(). This preserves default browser functionality for legitimate cases, such as cookies essential for same-site sessions or user preferences.

2. CapturePredefined: For cookies explicitly defined in the policy as private (e.g., session identifiers that must not leak cross-site), the cookie's value is embedded into the associated token by updating the token's value field. The token's access rights are updated to READ_WRITE to reflect that it now carries an active cookie value – essentially turning it into a final token. We do this for the reason how the additional utility on our tokens work (**??**). Instead of being stored in the browser's global jar, this updated token is forwarded to StageTokenForReturn(), a helper routine that serializes the token information and stages it for return to the originating application.

3. CaptureWildcard: Wildcard tokens behave similarly: if a domain is destined for the private jar by a wildcard rule, not only the cookie name but also the value is captured into the token. The only important difference is that here, the token's access rights are not updated (default is no access rights "NONE"), as otherwise the tracking library could simply read out the value again, echo it back to its server, and thereby circumvent the isolation.

4. Reject: If no token matches the cookie or if the policy forbids storing cookies for this domain, the cookie is just dropped. This prevents unwanted cross-site tracking by suppressing unauthorized cookie storage operations.

This ensures that all unmodified web behavior remains intact while Byetrack enforces policy-based restrictions transparently.

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 already exists in the header. This is simply done by constructing the cookie string "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 simply discard it as the app already has it. 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 extends Gecko's networking stack with a dedicated emission helper implemented in

HttpBaseChannel's EmitByetrackTokensToGeckoView(). This function is invoked from HttpChannelParent::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 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 mozillas JSON writer utility, the function iterate voer 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 side. 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 GeckoSession class – the same locaiton where LoadUri and similar events are processed. When the "GeckoView:Byetrack:FinalTokens" event is received, the Java handler constructs a ContentValues object containng the token JSON and writes it to the application's registered content provider. allowing it to update its local capability store.

4.4.3 Additional Utility

All additional utility functions are implemented in a separate ContentProvider exposed by the browser. In this content provider, we leverage the parameter "method" of the call function 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 (String)and EXTRAS (Bundle) 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. Only if the `canRead()` flag inside the payload is set does the provider return the corresponding cookie value as the field `"value"` in 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.

4.4.4 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 Custom Tab and TWA launches. Together, these components ensure that apps using standard AndroidX interfaces transparently benefit from Byetrack's protection without code modifications.

4.4.5 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() 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` 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 (e.g., in a multi-domain flow), the function ensures that tokens for these hosts are also included.

This injection step is completely transparent to the embedding app. Developers can use the same CustomTabsIntent or TrustedWebActivityIntent interfaces 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 (final tokens) encapsulate.

4.4.6 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. This ensures automatic token injection whenever an app uses CTs or TWAs.

Specifically, both CustomTabsIntent.launchUrl() and TrustedWebActivityIntent.launchTrustedWebActivity() were extended to call the ByetrackClient.attachTokens() method before invoking ContextCompat.startActivity(). 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. These allow developers (or future automation logic) to specify related domains that should receive tokens as well – for instance, if the application anticipates cross-domain requests as part of the same trust context.

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.

4.5 Integration into Existing HyTrack Demo Applications

Both HyTrack demo applications, `CrossAppLauncher` and `CrossAppTrackerOne`, originally launched Trusted Web Activities (TWAs) using Google Chrome's `android-browser-helper` library, 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 Google's `android-browser-helper`. Although Firefox does not natively support TWAs yet, no further changes to the applications were required: the TWA intents are transparently downgraded to standard Custom Tab intents within the browser.

4.6 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.

4.6.1 Securely Identifying the Calling Application

The most crucial challenge was securely identifying the application that launched a *Custom Tab*. 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, *Trusted Web Activities (TWAs)* are not affected by this problem, as they establish 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 (see Section 3.5).

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.

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`) and perform a challenge–response protocol during Custom Tab 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 Custom Tab 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 [5], 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 Custom Tab is later launched, the intent itself remains empty, and the browser uses the previously committed data.

While this design provides strong authenticity guarantees—since 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 [6]. In this approach, the Custom Tab intent is wrapped into a `PendingIntent` marked as `FLAG_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 without requiring major architectural changes to the Custom Tab launch flow.

4.6.2 Bridging Between Java and Native Layers

Another technical challenge concerned bridging functionality between the Java-based browser components (Fenix and GeckoView) and the native enforcement logic in the C++ layer. Although both parts share overlapping responsibilities, they are intentionally separated for security and maintainability reasons. However, this separation complicates cross-layer calls, particularly in contexts where no `GeckoRuntime` instance exists—such as during `ContentProvider` invocations.

Since launching a temporary runtime solely for token processing proved unreliable and introduced unnecessary complexity, we opted to strictly separate concerns: the Java layer handles token generation and management, while the C++ layer performs enforcement once the browser process and runtime are active. Both layers share a secret key to ensure secure coordination without requiring direct runtime invocations.

Chapter 5

Evaluation

5.1 Experimental Setup

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. Instead of installing the apps directly, we used our custom installer, which extracts and registers the policy with the browser before installation, enabling capability issuance.

Because the original HyTrack apps relied on the Android Browser Helper library–tailored for establishing Trusted Web Activities (TWAs) with Chrome–simply switching to our enhanced Firefox browser was insufficient. Therefore, we removed the Android Browser Helper dependency and established the required session connection to Firefox manually, allowing the apps to open TWAs (or Custom Tab 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 Custom Tab or Trusted Web Activity 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

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 Assessment of Primary Goals

To assess our mitigation's effectiveness, we follow the three primary goals established by the HyTrack authors for viable defenses:

1. **Support for Web Platform Features** Since our framework does not modify the browser's rendering engine, all standard web platform features (cookies, JavaScript, modern APIs) remain fully functional.

2. **Seamless Integration** No changes to the app's or browser's user interface were required. The mitigation operates transparently in the background, preserving a seamless user experience.

3. **Controlled Access to Shared Browser State** Our approach isolates untrusted domains defined as private while allowing trusted domains (e.g., SSO providers) to remain in the shared global jar. This ensures that legitimate cross-app use cases such as Single Sign-On (SSO) continue to function, while tracking across untrusted apps is prevented.

In summary, our mitigation achieves all three goals: (1) it maintains full web functionality, (2) it integrates transparently, and (3) it provides fine-grained control over shared browser state.

5.2.3 Developer Control and Transparency

Beyond mitigating tracking, our framework empowers developers with explicit control over cookie behavior. By declaring specific domains or cookie names as private in the policy, developers can prevent sensitive cookies from being stored in the global jar, ensuring that they remain confined to the app's local context. This enables transparent handling of embedded third-party content without risking leakage of sensitive cookies to external trackers. Additionally, developers can directly interact with issued capability tokens to read or modify cookie values from within their app, offering a fine-grained and auditable control model.

5.2.4 Compatibility with Existing Mechanisms

Our approach complements existing cookie isolation mechanisms, such as CHIPS [4], which is natively supported in Firefox (chapter 7). If a capability grants access to the shared jar, cookies are stored using the browser's native CHIPS partitioning. Otherwise, storage occurs entirely within the app's local context. Thus, our system extends rather than replaces existing isolation mechanisms, adding an additional layer of policy-driven isolation.

5.2.5 Backwards Compatibility and Integration Effort

When an app lacks a policy, our framework defaults to issuing an ambient capability that allows full access to the shared cookie jar, preserving backward compatibility with existing apps and browsers. If an app defines a policy but is launched in a browser without our modifications, the attached capabilities are simply ignored, and the app continues to function normally. This ensures graceful degradation and interoperability across browsers.

Integration effort for developers remains minimal. Replacing the browser library dependency and adding a simple policy file to the app's assets are sufficient steps. Launching TWAs or Custom Tabs remains unchanged from the developer's perspective, as all token-handling logic is performed transparently within the modified library.

5.2.6 Usability and Developer Experience

Our mitigation framework is designed to be unobtrusive and developer-friendly. It requires no new APIs, permissions, or configuration interfaces. Developers who wish to enable cookie isolation only need to include a policy file; those who do not can omit it and retain full backward compatibility. Developers might also only wish to only decouple their cookies from the browser state, for which they can simply mark their first party domain as private or use predefined capabilities entirely for specific cookies. This low-friction design ensures that the mitigation can be adopted incrementally without breaking existing workflows or user experiences.

Chapter 6

Discussion

Chapter 7

Related Work

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. 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.

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.

HyTrack [1] demonstrates a novel cross-app and cross-web tracking technique in the Android ecosystem by exploiting the shared cookie storage between Custom Tabs (CTs) and Trusted Web Activities (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 on Custom Tabs. Beer et al. [8] 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 – such as Single Sign-On or ad delivery – within mobile apps. This can be seen by taking a closer look at the two possible mitigation strategies discussed by the authors, namely Browser State Partitioning and Forced User Interaction. Modern browsers prominently adopt state partitioning to combat third-party tracking. Firefox's Total Cookie Protection (TCP) [9] and Safari's Intelligent Tracking Prevention (ITP) [10] both enforce per-site cookie jars, thereby limiting cookie-based cross-site tracking.

However, both approaches introduce significant drawbacks. 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. Google is actively working on a similar mechanism under the name CHIPS (Cookies Having Independent Partitioned State) [4]. 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 is the tracker itself. Our solution can be seen as extending this paradigm to the app level.

In contrast, Forced User Interaction avoids these problems by allowing the browser to use its shared cookie storage. 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.

This work addresses this gap by proposing a capability-based access control framework for Android applications using CTs and TWAs. By wrapping Cookies into fine-grained capability tokens – created by the browser according to the app developer's policy –, the browser decides which cookies are stored in the shared cookie jar and which are stored in the app-specific storage, depending on a flag analogous to CHIPS' *Partitioned* attribute. This ensures that there is no cross-app tracking possible for untrusted third-party libraries, as each app stores its own tracking cookie. As the shared cookie storage still exists for domains declared as first-party or trusted by the app developer, legitimate uses of the shared browser state (e.g. SSO) are preserved. Seamless integration of web content is also preserved, as there is no need for user interaction or changes to the UI. Thus, in contrast to prior discussed mitigation strategies, this approach provides developers with a practical and enforceable way to render cross-app tracking infeasible.

Our interpretation of capability tokens is inspired by JSON Web Tokens (JWTs) [3], 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. Our approach extends 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" [11] 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

Chapter 9

Conclusion

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

[3] 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

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

[5] 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].

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

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

[8] 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.

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

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

[11] 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

TODO

Use of Generative Digital Assistants

used Claude Sonnet Model 4.0 embedded in Visual Studio Code exclusively for understanding the firefox codebase and to help linking my code additions together.

Models like ChatGpt and Claude also used for debugging purposes (copy paste and let it try to fix the code or to explain obscure error messages).

...

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.

Downgrading Policy Example

Here, the predefined rule downgrades the predefined global access of *___royaleapi_session_v2* to the private predefined entry. Cross Conflicts on the other hand are ignored. The wildcard rule allows all cookies on the same domain to global storage, except the predefined one. Both entries are kept during downgrade.