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 Custom Tabs and Trusted Web Activities on Android

Custom Tabs (CTs) and Trusted Web Activities (TWAs) enable a seamless integration between apps and the web by sharing browser state – most notably, session cookies. While this enhances user experience, it also introduces a significant privacy risk: third-party libraries can exploit the shared browser state to track users across multiple apps in which they are embedded. For instance, HyTrack demonstrates that a single shared third-party library used across multiple apps is sufficient to persistently identify and track users, bypassing conventional browser and OS sandboxing mechanisms.

How are they launched under the hood? Where can attaching of metadata happen?

2.2 HyTrack Attack Overview

summarize its attack flow (with figure describing Hytrack?), key findings, and impact. Or should this go into its own chapter?

2.3 Root Causes of HyTrack and Our Mitigation Strategy

The feasibility of HyTrack relies on the following underlying weaknesses in the CT and TWA security model:

- **Implicit and Persistent Cookie Sharing**: All apps using CTs or TWAs inadvertently access a single, persistent global browser cookie jar, regardless of whether such sharing is intended by the developer. This cookie state persists across app launches and can survive typical user efforts to clear tracking data.

- **Lack of App Context in Browser**: The browser is unaware of which app initiated a given request, and therefore cannot enforce app-specific isolation or developer-defined policies. As a result, all apps using CTs or TWAs share the same cookie state, even if their purposes and trust levels differ.
- **Unrestricted Third-Party Inclusion**: Third-party libraries embedded in multiple apps gain access to the shared browser cookie jar, allowing them to identify users across app boundaries.

Our approach addresses these issues by introducing explicit, developer-defined policies and browser-enforced **capability tokens**:

- **Explicit Cookie Isolation**: Cookies are stored only if the domain is explicitly declared as trusted or untrusted in the app's policy, i.e. a capability for the domain exists. Based on the token(s) received by the app, the browser either stores the wrapped cookie in the shared cookie jar or returns it to the app for isolated local storage. This reverses the implicit sharing model and ensures that even persistent HyTrack identifiers cannot be used for cross-app tracking, as each app maintains a distinct, app-scoped identity – assuming the policy is correctly configured.
- **App-Aware Browser Context**: Each capability encodes an *App ID*, allowing the browser to enforce per-app cookie policies and prevent unintended delegation between apps. Additionally, capabilities include an *App Version Number*, enabling the browser to detect app updates and invalidate tokens that no longer reflect the current policy.
- **Capability-Scoped Access Control**: Malicious third-party domains are confined to app-local storage and cannot access the shared cookie jar, thereby blocking cross-app tracking via embedded libraries. At the same time, legitimate use cases, such as Single Sign-On (SSO), remain supported by granting trusted domains the necessary capabilities to access the shared cookie jar.

By making browser state access explicit, app-aware and scoped, our solution enforces cookie isolation across apps – particularly against embedded third-party libraries – thereby neutralizing HyTrack's core tracking mechanisms while preserving the seamless user experience of CTs and TWAs and supporting legitimate app-web integrations.

2.4 Capabilities

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

Chapter 3

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.

Chapter 4

Design and Methodology

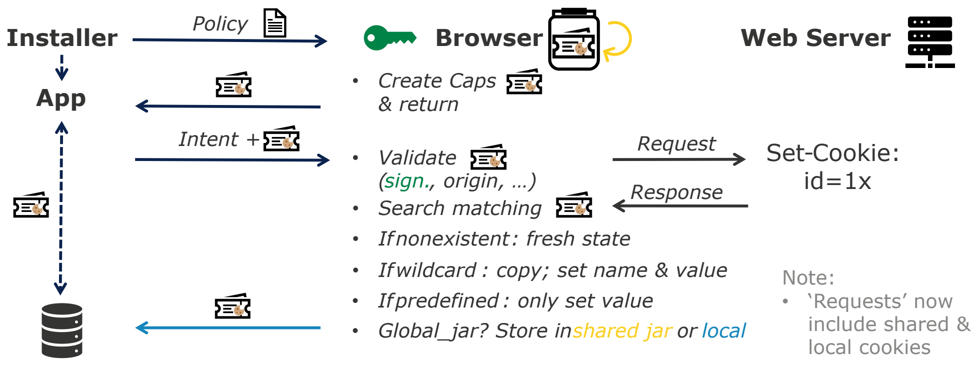

Installer
Policy
App
Intent +
Browser
Create Caps & return
Validate (sign., origin, ...)
Search matching
Ifnonexistent: fresh state
Ifwildcard : copy; set name & value
Ifpredefined : only set value
Global_jar? Store in shared jar or local
Web Server
Request
Response
Set-Cookie: id=1x
Note:
'Requests' now include shared & local cookies

Figure 4.1: High-level overview of the Byetrack flow between installer, app, browser, and web servers.

To mitigate cross-app tracking caused by shared browser state, we propose a developer-defined policy mechanism, combined with cryptographically enforced capabilities, to control how cookies are managed and isolated on a per-app basis.

4.1 The Policy

To benefit from the scheme, the app developer defines a policy specifying which domains should share browser state and which should isolate it. Optionally, the developer can also predefine specific cookie names expected from certain domains for more fine granular control. This granular control goas beyong the simple trusted/untrusted domain distinction, allowing developers to specify exactly which cookies should be shared and which should be isolated, especially useful in scenarios where the developers wants to embed own domains that might host both first-party and third-party content. If no policy is provided, the defense allows backwards compatibility by reverting to the default behavior of storing all cookies in the browser's shared jar. This way, the developer can choose to not implement any policy and still have the app function as before, albeit without the privacy benefits of the proposed mechanism.

4.2 The Capability Token Structure

Each capability token defines the following fields:

- **Cookie Name and Value:** Represent the actual cookie data to be stored and managed by the browser.
- **Signature:** Ensures the capability was issued by the browser and has not been tampered with.
- **Package Name:** Identifies the origin of the CT or TWA request, ensuring that only apps that were initially granted this token can make use it. This field is essential to prevent implicit delegation – without it, a receiving app could exploit the capabilities granted to the sending app and potentially gain unauthorized access.
- **Domain:** Specifies the designated destination web server, ensuring only capabilities issued for the domain launched will be considered to define the isolation scope for cookies received from that server. Otherwise, a malicious library included in the app could exploit capabilities issued for trusted domains to mislead the browser into storing cookies from untrusted domains in the shared global jar.
- **App Version Number**: Indicates the version of the app to detect potential policy changes and ensure consistency between app and policy. In case of a version mismatch, the browser rejects these capabilities. Every time the app is updated, the installer resends the policy to the browser, which can then issue new capabilities that overwrite the previous ones.
- **Rights:** Define the scope of access granted to the app –, whether it may request the browser to read the cookie name or value, or even write a cookie value. This restriction is essential to prevent libraries from exploiting browser access to extract capability values, which could otherwise be used for tracking, for example HyTrack that could use the cookie name as the identifier instead of the name.
- **Global Jar Flag:** Determines whether the shared cookie jar or an app-specific cookie jar should be used, i.e. whether the cookie should be send back to the app to be stored in its isolated jar or kept in the browser's shared jar.

We distinguish between *final* and *wildcard* capabilities. Final capabilitiies are fully specified, containing fixed cookie names and values, while wildcard capabilities leave these fields unspecified, allowing the browser to fill them in dynamically when cookies are received from the web server – thereby creating *final* instances of capability token. Therefore, the wildcard capabilities can be seen as templates for generating final capabilities on-the-fly, which fall into one of the following categories:

- **Classic Wildcard Capabilities:** The classic wildcard capability leaves wildcard fields for both cookie name and value. This token can either exist to state that all cookies from a specific domain should be stored in the global cookie jar or in the app-specific jar, depending on the global jar flag. Consequently, this means one of such token is sufficient per domain to either allow or disallow sharing of all cookies from that domain.
- **Predefined Capabilities:** Allows the developer to specify particular cookies by name are expected from a specific domain. This provides fine-grained control over cookie scoping and allows developers to isolate their app from the browser's state when desired, while still enabling seamless web-app integration.
- **Ambient Capability** Instruct the browser to revert to default behavior by storing all received cookies in the global cookie jar and including them in subsequent requests to the corresponding web server. These have the global jar flag set by default and augment a wildcard capability with the domain name, cookie name, and corresponding value.

  Note: The Ambient capability provides no security guarantees and serves solely as a fallback mechanism for maintaining functionality when no explicit developer policy is provided.

4.3 Capability initialization flow

During app installation, the installer extracts and transmits the policy to the browser. Before creating any capability tokens based on the received policy, the browser validates and, if necessary, downgrades the policy to ensure minimal privilege. This downgrade step prevents ambiguous or conflicting entries from resulting in overly permissive capability tokens. The downgrade follows these rules:

- Predefined conflicts: If a domain or cookie name appears under both predefined global and predefined private, the global entry is discarded and replaced by the private one. This ensures that private capabilities (i.e., those restricted to the app's private cookie jar) always take precedence over global capabilities.

- Cookie-level conflicts: If the same cookie name occurs in both global and private lists for the same domain, only the private entry is retained. This prevents a single cookie from being accessible with two different privilege levels.
- Wildcard conflicts: If the same domain appears in both wildcard global and wildcard private, the domain is downgraded to private. This ensures that domain-wide capabilities default to the least permissive (private) scope.
- Independence of wildcard and predefined sections: The predefined and wildcard sections are treated independently. This means a global predefined rule (e.g., for a specific cookie) and a private wildcard rule (e.g., for all other cookies on the same domain) can coexist. Likewise, a global wildcard and private predefined entry may both exist for the same domain. These combinations are allowed because they describe complementary rather than conflicting access scopes.

The result of this downgrade process is a sanitized, conflict-free policy that preserves legitimate combinations of predefined and wildcard entries, while eliminating overlapping or inconsistent privilege assignments. This ensures that capability tokens generated from the downgraded policy follow the principle of least privilege, minimizing the attack surface if an app misconfigures or intentionally inflates its declared policy. The generation process itself is straightforward: If the policy contains predefined entries, the browser generates corresponding predefined capability tokens for each specified domain and cookie name. If the policy contains only a domain stating that all cookies from this domain should be stored in the global jar, the browser creates a classic wildcard capability for this domain with the global jar flag set accordingly, and vice versa for the private jar. If no policy is provided, the browser creates an ambient capability with the global jar flag set, effectively reverting to the default behavior of storing all cookies in the shared jar. Finally, each token is signed, ensuring its authenticity and integrity before being encrypted to ensure the tracking library cannot even read its contents to conduct tracking by leveraging cookie names or values.

After successful generation, the browser sends all created wildcard capability tokens to the app, which stores them in its private wildcard storage for future use. The browser also lets the app know whether the app runs in ambient mode or not, i.e., whether a policy was provided by the developer or not. This is important for the app to know whether the token it received was the ambient capability or another token, as they are encoded and encrypted and thus indistinguishable for the app. Note: The app also stores the final capabilities separately in its private final storage, which are initially empty and may only be received upon requests to the web server.

Storing the wildcard and final tokens separately serves solely implementation convenience, as discussed in the implementation chapter 5.

When the app is updated, the installer resends the policy to the browser, which can then issue new capabilities that overwrite the previous ones.

4.4 CT and TWA workflow with our capability mechanism

When the app opens a URL via CT or TWA, the wilcard tokens (and up to this point empty final stokens) are attached to the respective intent used to start the activity. After successful decryption of the received capability tokens, the browser parses and verifies their authenticity by validating the signature, the app package name, version number and the destinatin domain. If any of these verifications fail, the browser discards the capability token and treats it as non-existent. Otherwise, the valid wildcard capabilities are used to (1) manage cookies received from the web server and the valid final capabilities are used to (2) construct a cookie header.

Upon receiving cookies, the browser matches them against the capabilities it holds for the browsing context provided by the app. The browser then processes each cookie with descending priority as follows:

1. If the token is ambient, the browser directly stored the cookie in the shared cookie jar, like it would do without any capability mechanism in place.
2. If a private predefined capability exists matching the name of the received cookie, the browser only fills in the matching cookie value in the capability and returns it to the app to store it in its isolated jar.
3. If the browser holds a private wildcard capability, both the cookie name and value are filled in accordingly and the capability is returned to the app for storage in its isolated jar.
4. If a global predefined capability exists matching the name of the received cookie, the browser stores the matching cookie in its shared jar.
5. If a global wildcard capability exists, any cookie received from the webserver has permission to be stored in the browser's shared jar.

6. If neither a predefined nor a wildcard capability exists, the browser discards the cookie and it is never stored.

When sending requests to the web server, the browser utilizes the valid final (in-app) capabilities received from the app to construct a cookie header. The browser merges the resulting cookie header with the cookie header created from the shared cookie jar by default and sends it to the web server.

The communication between browser and web server remains unchanged, i.e. the browser sends request and set-cookie headers as usual and the web server responds with a (customized) response and possibly new cookies.

4.5 Additional Utility

To enhance usability and developer experience, our mitigation framework includes additional utility that allow to get insights about the current final capabilities held by the app and thereby make use of the predefined capabilities. For this, the app can query the browser to (1) get the names of the cookies encapsulated in the final capabilities, (2) retrieve the cookie values of final tokens and (3) write/update cookie values in final tokens. The browser only grants these requests if the corresponding rights are granted in the capability token, that is read rights for (1) and (2) and write rights for (3).

4.6 Benefits

Beyond eliminating cross-app tracking through HyTrack and its postulated goals, we identify the following additional benefits offered by our approach:

B1) **Fine-Grained Control**: Developers can specify which cookies are shared and which are isolated, allowing for a more tailored approach to privacy.

B2) **No Browser State for Apps**: The browser is stateless with respect to app-specific capabilities, as these are retained and transmitted by the app with each intent invocation.

B3) **No Third-Party Code Changes**: The web server does not need to be aware of the capabilities or make any changes to its code, as the browser handles the capability management.

B4) **Backwards Compatibility**: In case the developer does not implement its own policy, it will default to allowing any domain using the shared browser state, consequently behaving like the state-of-the-art cookie management.

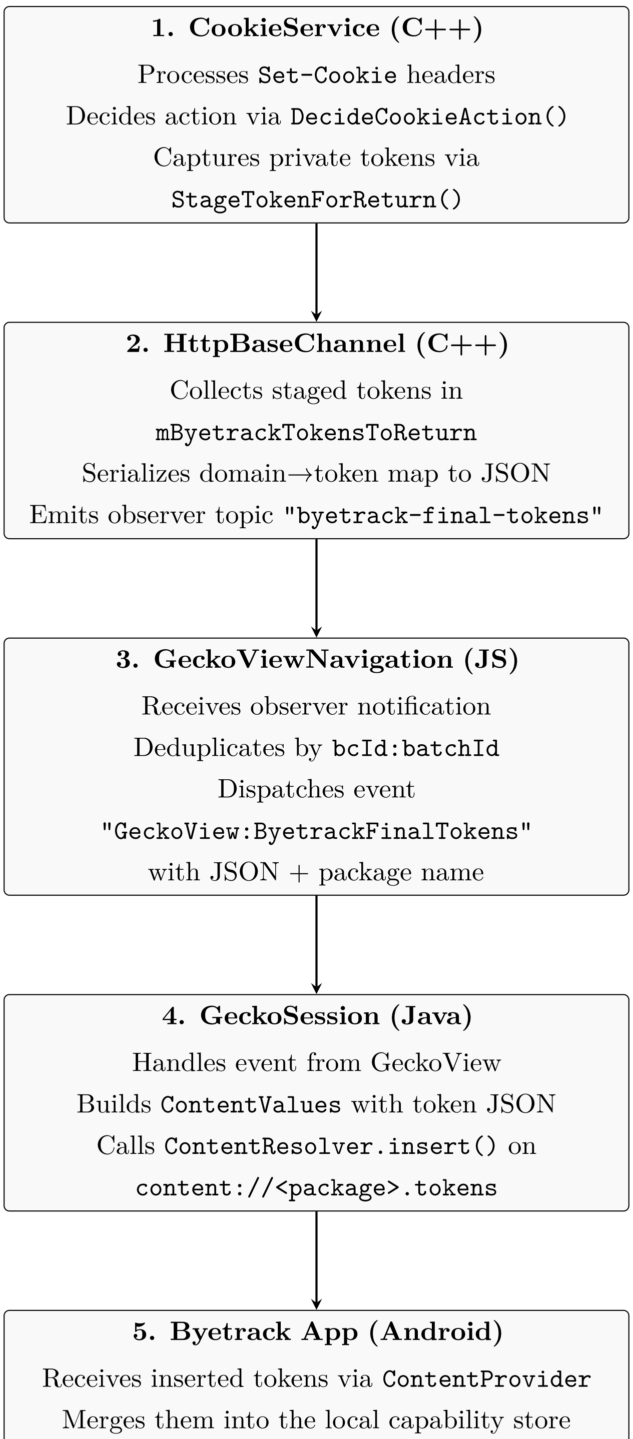
4.7 Other Considerations

Instead of the installer sending the app's policy to the browser, which then generates the capabilities accordingly and sending them to the apps, an alternative approach would be to let the installer directly generate the capabilities based on the policy and send them to the app.

This would free the browser from the resposibility of generating the capabilities and thus reduce its complexity to only enforcing them and would no longer require communication between installer and browser for each app. For further deployment, this could be beneficial as the installer is part of the OS and thus does not require updates like the browser would, such as only require cooperating browsers to only implement the enforcemt. However, this would require a decentralization of the token functionality and thus require the installer and browser to initially share a secret key, which increases the trusted computing base and thus the attack surface. Therefore, we decided against this approach and chose the browser as the sole trusted component responsible for capability generation and enforcement for our proof-of-concept.

Chapter 5

Implementation



1. CookieService (C++)
Processes `Set-Cookie` headers
Decides action via `DecideCookieAction()`
Captures private tokens via
`StageTokenForReturn()`

2. HttpBaseChannel (C++)
Collects staged tokens in
`mByetrackTokensToReturn`
Serializes domain→token map to JSON
Emits observer topic `"byetrack-final-tokens"`

3. GeckoViewNavigation (JS)
Receives observer notification
Deduplicates by `bcId:batchId`
Dispatches event
`"GeckoView:ByetrackFinalTokens"`
with JSON + package name

4. GeckoSession (Java)
Handles event from GeckoView
Builds `ContentValues` with token JSON
Calls `ContentResolver.insert()` on
`content://<package>.tokens`

5. Byetrack App (Android)
Receives inserted tokens via `ContentProvider`
Merges them into the local capability store

To demonstrate the feasibility of our mitigation strategy, we developed a proof-of-concept installer application, a new library that ships the changes and modified the androix browser library to inject our capability tokens for each call to launch a Custom Tab (CT) or Trusted Web Activity (TWA).

In this proof-of-concept, we chose the firefox mobile browser (Fenix) to act as the policy enforcement point, but it could have been any other browser the authors of HyTrack [1] found out to be vulnerable to their attack, such as Chrome or Brave.

We modified the proof-of-concept apps provided by the authors of HyTrack and provide a test application for more insight into the framework's behavior. For completeness, we also provide a "evil" acting app that demonstrates what the HyTrack library included in tan app could do to circumvent the mitigation.

Put a diagram here that shows the components and their interactions!!!

5.1 Policy Format

The Byetrack policy defines the capability configuration that determines which domains are eligible to receive capability tokens and under which isolation context (global or private) they operate. The policy is expressed in a structured JSON format that is divided into two primary 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 specifiec 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 (usable across trusted contexts) and a private token (restricted to one context). 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 explaination can be found in the appendix 5.

5.2 Custom Installer

The installer stored the apks of the apps to be installed in its assets folder. This folder is read-only at runtime, so we copy the apks to the app's private storage first when an installationis initiated. The apks uri is then installed by setting it in the intent, wich is finally launched. As we can only read the policy with the help of the AssetManager once the apk is actually installed, we use ActivityForResult to directly continue after the installation is finished to circumvent the need to continuously poll for the installation status. Once the policy json file is read into a json string, it is sent to the browser together with the installed apps version and package name provided by the PackageManager by calling a designated content provider exposed by the browser. The reason for choosing a content provider for inter process communication is that it is easy to implement by just extending the ContentProvider class and registering it in the manifest and offers functionality to receive the identity of the calling app, which is crucial for our use case. Even though in latest Android versions (CITE HERE), there exists functionality to get the calling package name of an intent received via a broadcast, it is not reliable and in my case only returned null. Attaching the policy, package name and version as extras to the intent and wrapping it in a pending intent is a workaround, but this faces the problem of spoofing, as the tracking library could simply create the pending intent itself and send a fake policy to the browser. A service could be another option, but it is more complex to implement and requires more boilerplate code, even though being the more clean solution by establishing a reusable communication channel between the installer and the browser. The downside of using a service is that the browser needs to be running in order for the installer to connect to it, which in this scenario is not guaranteed.

5.3 Browser (Fenix)

The modifications conducted in the browser can be divided into two main parts: token generation and additional utility in the java layer and the actual enforcement in the C++ layer. Despite the overlapping functionality, it was best to stricly isolate the two parts and have them "share" a secret key. This is due to the fact that once the content provider is called, the browser is not running and therefore no GeckoRuntime exists. Having no runtime though makes it impossible to call into the C++ layer directly from the content provider. Trying to launch a temporary runtime to call into the C++ layer – despite it introducing a lot of complexity and overhead – turned out to be troublesome and error prone as it messed with the actual runtime of the browser once it was started.

5.3.1 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 tokens are ever created. Note that the cooki name parameter is set to "*" for wildcard tokens to indicate that they apply to all cookies for the given domain and that the rights parameter is set to "NONE" for all wildcard tokens. 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 (5.4).

5.3.2 Launching Custom Tabs / TWAs with Tokens

As our goal is to isolate cookies based on our capability tokens, and cookies sending/receiving is handled in the C++ layer (in CookieService) of firefox's browser engine Gecko, we first need to thread the tokens down from the java layer through the JNI bridge into the C++ layer.

5.3.2.1 Threading

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, and returns them as a structured map.

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. At this point, the capability tokens and caller metadata become available to the internal Gecko loading pipeline.

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 DocumentLoadListener 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.

5.3.2.2 Enforcement / 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.

(1) The main flow for attaching cookies to outgoing requests happens in HttpBaseChannel's AddCookiesToRequest function. Here, the CookieService's GetCookieStringFromHttp is used to retrieve the appropriate cookies for the request's target domain. Similarly, we use the top level browsing context to retrieve our prebuild Cookie header string based on the final capabilities associated that were attached to the top level browsing context during the document load phase. The outgoing request's header is then simply built by appending our capability-based Cookie string to the existing Cookie header separated by a semicolon.

(2) For incoming Set-Cookie headers, the HttpBaseChannel's SetCookieHeaders function is responsible for processing and storing cookies received from server responses. It calls the CookieService's SetCookieStringFromHttp function iteratively for each cookie string found in the response headers storing them in the browser's cookie jar. Here, we again leverage the top level browsing context to retrieve the wildcard capability tokens and pass them on via an additional parameter to the SetCookieStringFromHttp function. As we need to check the tokens for each cookie individually, there is no shortcut like in the previous case and we need to change the behavior of how cookies are stored based on the wildcard capabilities. This function is also the place where the CHIPS [4] implementation in Firefox is located, so it was straightforward to add our own logic here. In fact, the CHIPS logic is executed first, and only if it allows the cookie to be stored, our Byetrack logic is applied 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. This function evaluates all available tokens for the current cookie 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 actual token is again serialzed 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 containg the token JSON and writes it to the application's registered content provider. allowing it to update its local capability store.

5.3.3 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.

5.4 App-side changes

On the application side, two main components were introduced to integrate the Byetrack mitigation seamlessly into existing Android apps: (1) a standalone Byetrack helper library, which encapsulates all logic related to capability token management and injection, and (2) minor modifications to the AndroidX Browser library that transparently hook into the Custom Tabs and Trusted Web Activity launch mechanisms. Together, these ensure that any app using the standard AndroidX Browser interface automatically benefits from the Byetrack functionality without requiring manual integration.

5.4.1 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() 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.

5.4.2 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. 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.

5.5 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.

5.6 Implementation Challenges

5.6.1 Securely Identifying the Calling App

The most challenging and crucial part of the implementation is identifying the app that launched the Custom Tab in a secure manner. Note this does not apply to TWAs, as they are required a session to be established with the browser first (binder channel), which already provides the app identity in a secure manner.

This is the most important piece of information, as it is needed to verify that the capability tokens presented to the browser actually belong to the app that launched the Custom Tab. Therefore, it is essential that this information cannot simply be spoofed by a third-party app. Even though Android's Intents form a IPC channel between apps, they do not provide built-in functionality to securely identify the calling app via the intent itself.

We tried multiple approaches to solve this problem, where most of them fail due to being insecure regarding our thread model 3 or because they require modifications that would change a simple CT launch into a overly complex flow. Here are some of the approaches we tried:

Plain Intent Extras: The simplest approach is to attach the app's package name as extra to the intent in the same way as we do with the capability tokens so that the browser can read it out. However, this approach is fundamentally insecure, as a package name is just a String and thus any other app could simply attach the same package name to the intent and thereby impersonate another app.

Shared Secret and Intent Extras: The next idea was to share a secret key between the app and the browser during installer via for example `KeyStore` and establish a challenge-response protocol. This fails for the reason that we have to assume that the tracking library included in the app can also access the shared secret, as it has the same permissions as the app itself. With this knowledge, the tracking library could simply do the challenge-response itself and share information with other collaborating apps, thereby circumventing the approach.

Using Medium that provides Caller Identity: Another idea was to fall back to a medium that already provided the caller identity in a secure manner from which the browser could derive the package name and link it to the intent launching the Custom Tab. This turned out to be quite difficult, as it is hard to use a identifier the app can send to the browser that cannot be read by the tracking library and therefore spoofed.

Commitment Scheme: An approach that follows this idea is inspired by commitment schemes, which by design features a two-phase protocol: commit and reveal [5]. First, the app would commit the requested url and capability tokens for launching a CT to the browser in advance via a secure channel from which the app identity can be derived. Then next time the app launches a CT, the Intent would remain "empty" without any extra data and the browser would launch the CT based on the previously committed data.

With this approach, the tracking library would not be able to spoof the app identity, as the commit phase happens over a secure channel. Despite being able to make its own commitments, the tracking library is limited to the app's identity and therefore the tokens it holds.

However, there are multiple downsides to this approach: Instead of only launching a CT with a single intent and injecting the tokens there, the app now has to perform two separate operations, which increases complexity and overhead. Also, any other app could launch a CT based on previously committed data, as there is no link between the commit and reveal phase.

Pending Intent: A similar approach that overcomes the downside of having two separate operations is to use Android's `PendingIntent` mechanism [6]. The CT intent could be wrapped into a PendingIntent flagged as immutable for launching the CT, the tokens included as additional data before being sent over to the browser. The browser could then retrieve the package name by calling getCreatorPackage on the PendngIntent object and invoke it to launch the CT on behalf of the app.

Other apps would still be able to launch pending intents of different apps if shared, but they would not be able to spoof the app identity and thereby impersonate another app and leverage its tokens.

5.6.2 Other Challenge

What else?

Chapter 6

Evaluation

To assess the effectiveness of our proposed mitigation strategy, we adopt the three primary goals identified by the authors of HyTrack as essential for any viable defense:

1) **Support all features of the web platform:** The solution must allow applications to display fully functional web content, including support for cookies, JavaScript, and modern APIs.
2) **Preserve seamless integration:** The user experience must remain uninterrupted. This includes avoiding obtrusive permission dialogs and maintaining smooth transitions between native and web content.
3) **Enable controlled access to shared browser state:** While isolation is required to prevent cross-app tracking, legitimate scenarios such as Single Sign-On (SSO) must remain functional.

These criteria reflect the fact that HyTrack exploits standard Android behavior – specifically, the shared browser state exposed through Custom Tabs and Trusted Web Activities – rather than relying on unauthorized access or system vulnerabilities. Therefore, naive approaches like disabling shared cookies entirely would break common use cases and are not acceptable.

To validate these hypotheses, we will build on the open-sourced measurement tooling and proof-of-concept applications provided by the authors of HyTrack. Specifically, we plan to:

- Replicate the original HyTrack experiments under controlled conditions using two unrelated Android apps that embed the tracking library (similar to the HyTrack demo).

- Apply the mitigation framework and compare observed behavior against the baseline.

We will collect and analyze the following metric:

- Number of Capabilities created and used by the browser.

In doing so, we aim to demonstrate that the proposed solution effectively blocks HyTrack's cross-app tracking channel while preserving compatibility with existing web features and maintaining seamless user experience. Additionally, we will show that this strategy can be used by developers to control cookie transmission in a fine-grained and policy-driven manner to enable safer and more transparent integration of third-party web content.

6.0.1 Test Setup

TODO

6.0.2 Test App

TODO

6.0.3 Results

TODO

Chapter 7

Discussion

Chapter 8

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 9

Future Work

Chapter 10

Conclusion

Chapter 11

Open Science

To promote transparency and reproducibility in our research, we have made all relevant artifacts publicly available. This includes the source code for our mitigation framework, the modified fenix browser and a test application demonstrating the functionality, next to the by HyTrack proof-of-concept applications instrumentalized with our mitigation.

SOMETHING ABOUT LICENSES?

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.