**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.

## 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

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

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

The modifications conducted in the browser can be divided into two main parts: token generation and additional features 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.

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 Enforcement

On receiving the intent, the browser verifies the signature, validates that the app ID and version match, and enforces the described scope and rights when deciding how to store cookies.

### 5.3.3 Additional Features

## 5.4 Libraries

### 5.4.1 Byetrack/Mitigation

### 5.4.2 AndroidX Browser

## 5.5 HyTrack Demo Apps

## 5.6 Test App

## 5.7 Evil App

# 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.

# 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 [4] 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. [5] 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) [6] and Safari's Intelligent Tracking Prevention (ITP) [7] 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) [8]. 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) [**?** ], 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" [9] 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] LocalLeaks, "Tracking users with localhost: Facebook's covert redirect abuse," https://localmess.github.io/, 2023.

[5] 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.

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

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

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

[9] 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.