Evaluation of Security and Privacy Flaws of Cross-platform Android App Development Frameworks

BY

FEDERICO CIVITAREALE
B.S, Politecnico di Torino, Turin, Italy, 2023

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2025

Chicago, Illinois

Defense Committee:

Jason Polakis, Chair and Advisor
Xiaoguang Wang
Fulvio Valenza, Politecnico di Torino

ACKNOWLEDGMENTS

First of all, I want to thank my family, without whose support this experience wouldn't have been possible. To my parents, who allowed me to stay here without ever asking for anything in return, and to my sister, whom I always feel I can count on.

I'd also like to express my gratitude to my professor and his team, who helped me accomplish this work. You were there when I had doubts or needed to solve problems.

I'm glad to have met so many special people in Chicago, who helped make this experience even greater. A special thanks to Riccardo, companion of a thousand adventures, and to Francesco, with whom I shared endless gym sessions.

I'm grateful for my friends overseas, Riccardo, Giorgio, Fabio, Alessandro, Luca, Samuele, Ludovico, Lorenzo e Alessandro who despite the distance were always available for a chat or when I needed to vent.

Last but not least, I want to thank Lucía, who is my safe haven after hard days of work, and who brought out a part of me that I didn't think I had. No importa si estás cerca o lejos, siempre te estaré agradecido. Te quiero.

FC

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

UIC	University of Illinois at Chicago
CSP	Content Security Policy
CSP	Same-Origin Policy

SUMMARY

Summary

In recent years, the adoption of cross-platform frameworks for mobile application development has grown substantially, driven by the need to shorten development cycles, reduce maintenance costs, and enable the deployment of applications across multiple platforms from a shared codebase. Frameworks such as *Cordova*, *Ionic*, and *React Native* allow developers to build Android applications using familiar web technologies (HTML, CSS, and JavaScript), which are then executed within native containers on mobile devices. Although this model provides significant advantages in terms of portability and productivity, it also introduces security and privacy risks that are not typically present in applications developed natively.

The risks associated with hybrid applications mainly arise from their architectural structure. These applications involve continuous interaction between a web layer, responsible for the user interface and some application logic, and a native layer, which provides access to system features and hardware resources. Communication between these layers occurs through JavaScript bridges that, if misconfigured or not adequately protected, may allow untrusted code to obtain elevated privileges or access sensitive data. Furthermore, practices commonly used in web development, such as remote script inclusion or the absence of a strict Content Security Policy (CSP), can become significantly more dangerous in a hybrid environment where web code is executed outside of the browser's traditional sandbox.

SUMMARY (continued)

This thesis presents a systematic analysis of the security and privacy implications of cross-platform Android frameworks. To support the evaluation, a dataset of Android applications was collected from public sources, including the AndroZoo repository and platforms such as APKMirror and APKPure. The applications were decompiled and subjected to static analysis using a set of automated scripts developed explicitly for this purpose. These scripts are capable of detecting insecure configurations, dangerous permission exposures, and known vulnerability patterns related to WebView usage and native plugin integration.

The methodology enables scalable and reproducible testing across different application builds and framework versions. Preliminary results indicate that hybrid applications frequently expose a broader attack surface compared to their native counterparts. The most common issues identified include permissive WebView settings, missing or misconfigured CSP policies, unsafe use of JavaScript interfaces, and reliance on external or dynamically injected code.

The findings underscore the need for increased awareness and more stringent security practices in the development of hybrid applications. Additionally, the analysis and tools developed in this work contribute to the improvement of automated vulnerability detection techniques and provide a foundation for future research focusing on dynamic analysis and the identification of emerging threats in cross-platform mobile environments.

CHAPTER 1

INTRODUCTION

1.1 Context and Motivation

Over the past decade, mobile applications have become a central component of modern digital life. Smartphones are now used not only for communication and entertainment, but also for banking, identity management, health monitoring, and professional collaboration. As a result, mobile applications must be reliable, constantly updated, and available across multiple platforms. This growing complexity has pushed developers and organizations to seek solutions that allow faster development cycles and reduced duplication of effort.

Cross-platform development frameworks emerged in response to this need. Instead of maintaining separate codebases for Android and iOS, these frameworks enable the writing of most application logic once and its deployment across multiple platforms. Among the most widely adopted solutions in this field are *Cordova*, *Ionic*, and *React Native*. Each of these frameworks builds upon familiar web technologies such as HTML, CSS, and JavaScript, enabling developers with a web background to transition into mobile development with relative ease.

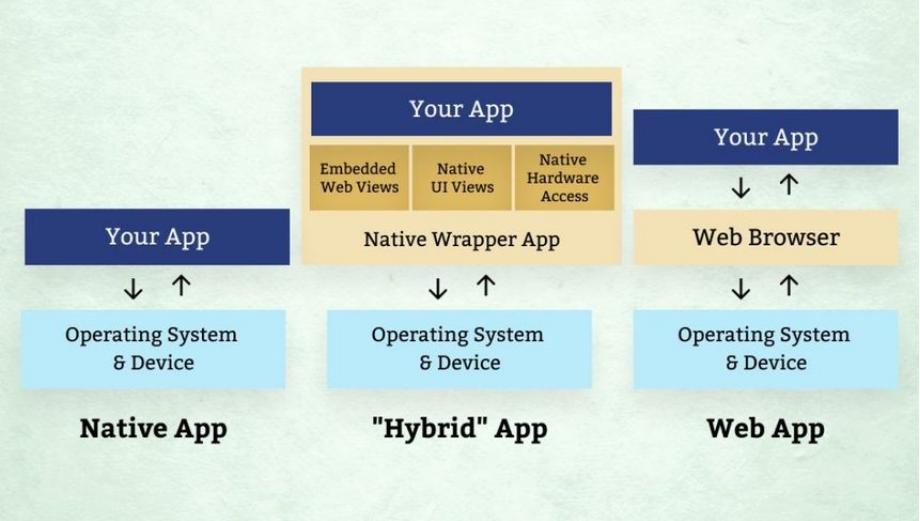

Figure 1: Comparison between native, hybrid, and web mobile application architectures.

Although the motivations behind cross-platform frameworks are practical and compelling, their architecture raises significant considerations regarding security and privacy. Hybrid applications typically operate at the intersection between two distinct execution environments: the web environment, which was initially designed to run inside the confined and sandboxed space of a browser, and the native environment of a mobile operating system, which has access to device-level resources such as storage, sensors, and network interfaces.

1.2 Problem Statement

The interaction between these two layers is mediated through communication bridges that expose native functionality to JavaScript code. While these bridges are essential for enabling hybrid applications to function, they also represent potential attack surfaces. If not carefully configured, they may allow untrusted web content to trigger privileged operations, bypass plat-

form security controls, or access sensitive user data. Furthermore, standard web development practices, such as relying on external content sources, dynamically injecting scripts, or omitting a strict Content Security Policy (CSP), may introduce vulnerabilities that have far more severe consequences in a hybrid mobile context than in a traditional browser environment.

Despite the increasing adoption of hybrid development approaches, many developers may not be fully aware of these risks, and there is still limited systematic evidence regarding how widespread certain insecure configurations are in real-world applications. This gap forms the central motivation of this thesis.

1.3 Objectives and Contributions

The main aim of this thesis is to examine the security and privacy implications of cross-platform mobile development frameworks, focusing on Android applications developed using Cordova, Ionic, and React Native. To achieve this goal, the work involves collecting and analyzing real-world hybrid applications, identifying recurring insecure patterns, and proposing a reproducible methodology for large-scale security evaluation.

To support this study, a dataset of Android applications built using Cordova, Ionic, and React Native was collected from publicly available sources, including the AndroZoo repository and APK distribution platforms such as APKMirror and APKPure. The applications were decompiled and analyzed using a set of automated scripts explicitly developed for this work. These scripts are designed to detect configurations, interfaces, and code patterns known to introduce security weaknesses in hybrid application environments.

The main contributions of this thesis are as follows:

- An overview of the architectural characteristics and security implications of the Cordova, Ionic, and React Native frameworks.
- The development of automated analysis scripts capable of detecting insecure patterns and configurations in decompiled Android applications.
- The collection and evaluation of a large dataset of real-world applications, enabling reproducible and scalable assessment.
- The identification of recurring vulnerability trends and developer practices that contribute to increased exposure in hybrid applications.

1.4 Thesis Structure

The remainder of this thesis is structured as follows.

1. **Chapter 2: Background and Related Work** - introduces essential concepts regarding mobile application architectures, explains the differences between native, web, and hybrid apps, and provides an overview of the Cordova, Ionic, and React Native frameworks. It also presents relevant security mechanisms and summarizes prior research on vulnerabilities in hybrid apps.
2. **Chapter 3: Methodology** - outlines the approach used for collecting samples of Android applications, implementing the automated static analysis script, and utilizing Frida as a dynamic analysis tool.
3. **Chapter 4: Experimental Findings** - presents the results of an exploratory vulnerability analysis conducted on controlled mock applications, describing the security issues

identified in each framework and explaining how these observations influenced the design of the automated detection logic. Additionally, a real-world application is exploited to illustrate these concepts.

4. **Chapter 5: Evaluation and Results** - presents the experimental findings of the study, highlighting common misconfigurations, vulnerabilities, and framework-specific risks observed across the analyzed applications.

5. **Chapter 6: Conclusion** - summarizes the primary outcomes of the research and suggests directions for future studies and improvements in automated analysis techniques.

CHAPTER 2

BACKGROUND

2.1 Mobile Application Models

The rapid evolution of mobile computing has given rise to several distinct approaches for building mobile applications. Each approach differs in terms of the technologies used, the performance trade-offs, and the level of access to system resources. Understanding these models is essential for contextualizing the security and privacy challenges discussed in this thesis.

Broadly speaking, mobile applications can be categorized into three main models: **native**, **web-based**, and **hybrid** applications. More recently, a fourth model, represented by modern cross-platform frameworks such as *React Native*, has emerged, combining aspects of both hybrid and native paradigms.

2.1.1 Native Applications

Native applications are developed specifically for a particular operating system, such as Android or iOS, using the platform's official programming languages and software development kits (SDKs). On Android, this typically involves Java or Kotlin combined with the Android SDK, while iOS applications are developed using Swift or Objective-C with Xcode.

Native applications offer direct access to device capabilities, including sensors, storage, network interfaces, and other system-level APIs. This allows them to deliver high performance and a seamless user experience that aligns closely with the look and feel of the underlying

platform. However, native development requires maintaining separate codebases for different operating systems, which can significantly increase development time and cost.

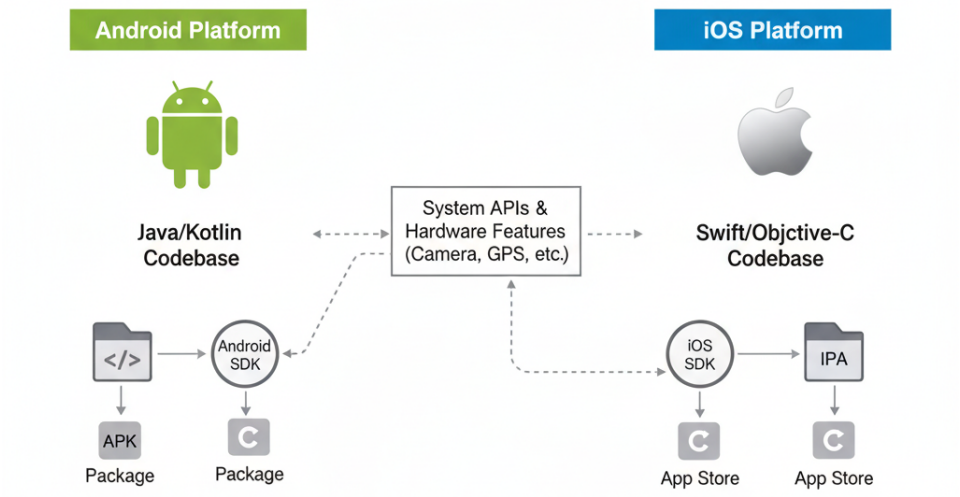

Figure 2: Native application model.

2.1.2 Web Applications

Web applications, by contrast, are built using standard web technologies - HTML, CSS, and JavaScript - and executed within the browser environment. They are platform-independent by nature, as the browser acts as a runtime layer that abstracts away the underlying operating system.

Because web applications run in a sandboxed environment, they have very limited access to system resources. This model greatly enhances user security but also limits functionality,

especially for use cases that require integration with native device features such as camera access, Bluetooth communication, or local storage beyond the browser's scope.

Web applications are easy to deploy and update, as changes can be made directly on the server without requiring users to install new versions. However, their dependency on a network connection and restricted access to hardware features limit their suitability for more complex mobile applications.

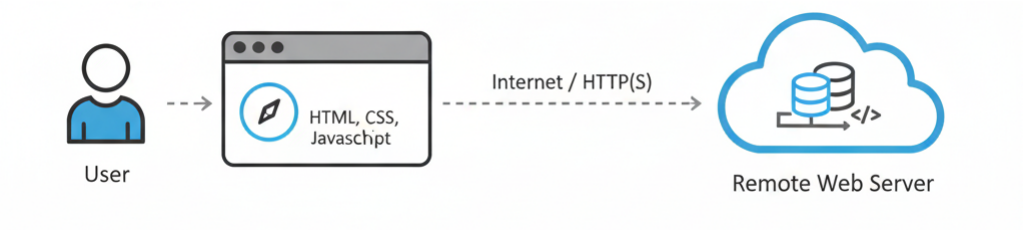

Figure 3: Web application model.

2.1.3 Hybrid Applications

Hybrid applications aim to combine the portability of web technologies with the broader access to system resources available in native apps. They are built using web technologies (HTML, CSS, JavaScript) but are packaged inside a native container that includes a component called the *WebView*. This WebView acts as an embedded browser within the app, capable of rendering web content locally while still providing a bridge to the native APIs.

This approach enables developers to reuse a large portion of their codebase across platforms while still distributing their applications through standard app stores. Frameworks such as *Apache Cordova* and *Ionic* exemplify this model. These frameworks provide a set of plugins that expose native functionality - such as camera, geolocation, or file system access - through JavaScript interfaces.

While this model offers clear development advantages, it also introduces security concerns. The interaction between the web and native layers creates a boundary where untrusted code could potentially invoke privileged operations. The way this bridge is implemented and secured plays a central role in the vulnerability landscape of hybrid applications.

Figure 4: Hybrid application model.

2.1.4 Cross-Platform Native Frameworks

In recent years, frameworks such as *React Native* have emerged to address some of the performance limitations of traditional hybrid frameworks. Unlike Cordova or Ionic, React Native does not render its interface inside a WebView. Instead, it uses JavaScript to control native components directly through a bridge that communicates with the underlying platform's UI elements and APIs.

This architecture provides a middle ground between hybrid and native development: it allows developers to use a single JavaScript codebase while still producing applications that perform nearly as well as fully native ones. However, the introduction of a JavaScript bridge still poses potential security challenges similar to those in hybrid apps, especially when third-party modules or dynamic code execution are involved.

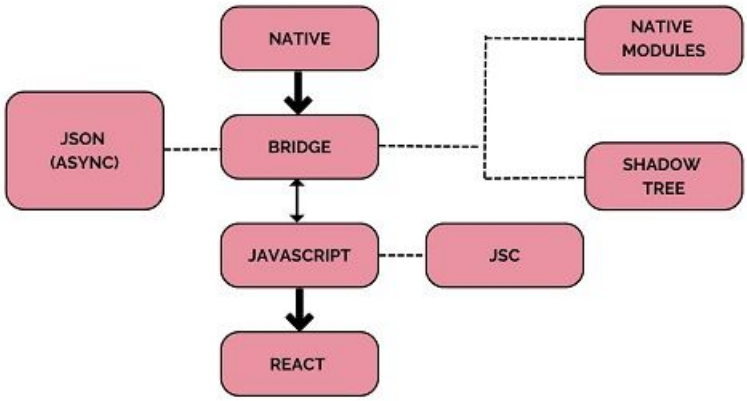

Figure 5: React Native application model.

2.2 Cordova Framework

Apache Cordova is one of the earliest and most influential frameworks for cross-platform mobile development. Originally created under the name *PhoneGap* in 2008, it was later donated to the Apache Software Foundation and renamed to Cordova. The framework became a cornerstone of hybrid app development, providing the foundation upon which several other frameworks - including Ionic - were later built.

Cordova enables developers to create mobile applications using standard web technologies such as HTML, CSS, and JavaScript. These web resources are bundled into a native application package, typically an Android APK, and executed inside an embedded component known as the *WebView*. From the user's perspective, a Cordova app appears indistinguishable from a native application, but internally it renders and executes its interface within this WebView container.

2.2.1 Architecture Overview

The architecture of Cordova can be understood as a layered model that integrates web and native components. At the core, the framework provides a bridge between JavaScript code running in the WebView and native APIs implemented in Java or Kotlin on Android. This bridge is realized through a bidirectional communication channel that allows JavaScript code to invoke native functionality and receive responses asynchronously.

When an application requests access to a native feature (for example, the camera or file system), the JavaScript code calls a specific API exposed by a Cordova *plugin*. This plugin translates the request into native code, executes the corresponding operation on the device, and

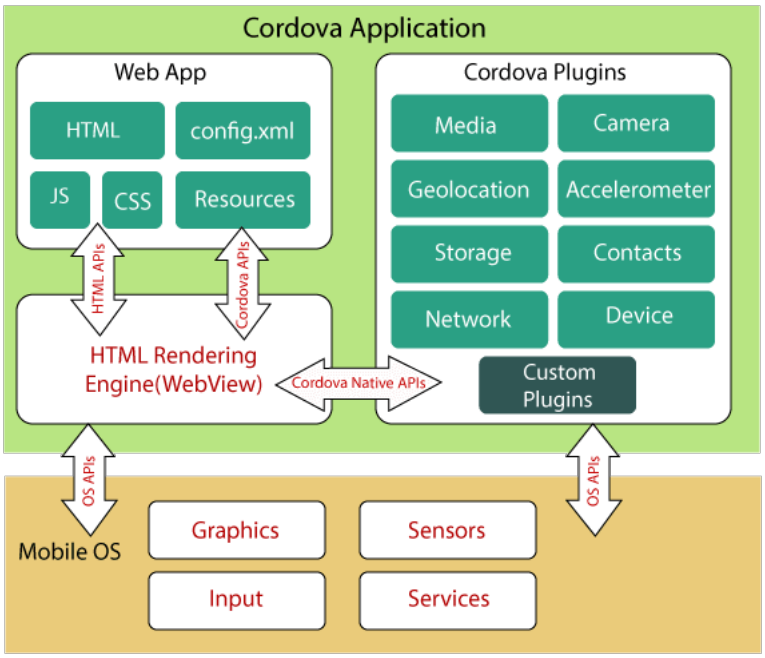

Figure 6: Apache Cordova architecture.

then returns the result to the JavaScript environment. This mechanism allows web-based code to perform actions that would normally be restricted in a standard browser environment.

Navigation Policies and the Role of CSP

During the exploratory testing phase, it became clear that Cordova's navigation configuration introduces additional complexity that directly impacts the application's security posture. Cordova defines three main navigation-related directives: `allow-navigation`, `allow-intent`, and `access-origin`. Although these settings appear to control the same aspects of external resource loading, they operate at different levels of the security model, and only the Content Security Policy (CSP) provides true enforcement of script-level restrictions.

`allow-navigation` determines which external URLs can be loaded *inside the WebView*. If misconfigured, for example, by using a wildcard such as `"*"`, the WebView may load untrusted content that executes with full access to the Cordova JavaScript bridge.

`allow-intent` regulates which external applications can be invoked via Android intents. While less directly related to script injection, overly permissive intent filters can facilitate phishing or malicious app redirection attacks.

`access-origin` is intended to control which external domains the application may contact via XHR requests. However, this directive is largely superseded by modern browser security mechanisms and is *not recommended* for use. In practice, it encourages developers to take a permissive approach (e.g., using `*`), which weakens the isolation between local `file://` resources and external domains.

Despite these configuration mechanisms, it is the **Content Security Policy (CSP)** that ultimately governs whether external scripts can execute within the WebView. Even if `allow-navigation` permits loading a remote page, CSP determines whether that page can run JavaScript, load additional scripts, or interact with unsafe inline code. As confirmed during testing, CSP is the strongest and most reliable defense against external script injection in Cordova, whereas navigation directives merely control resource loading without preventing code execution.

The divergence between Cordova's configuration directives and CSP enforcement explains why many real-world Cordova applications are vulnerable to injection-based attacks: developers frequently configure navigation policies permissively without understanding that CSP is the

actual mechanism responsible for restricting script execution. For this reason, CSP must be defined carefully, and directives such as `access-origin` should be avoided entirely.

TABLE I: SECURITY IMPACT OF CSP VS. ALLOW-NAVIGATION SETTINGS.

Scenario	**Can script run in CURRENT App?**	**Can App navigate to malicious.com?**	**Can script run on malicious.com?**
No CSP	**YES** (Vulnerable)	**YES** (If `allow-navigation` is `*`)	**YES** (Because `malicious.com` controls the rules there)
Strict `allow-navigation`	**YES** (If CSP is missing)	**NO** (Blocked by Cordova)	**NO** (Page never loads)
Strict CSP	**NO** (Blocked by CSP)	**YES** (Unless `allow-navigation` blocks it)	**NO** (Because the XSS couldn't force the redirect in the first place)

WebView Navigation Modes

Cordova applications rely on a specific navigation model that determines how URLs are opened when a link is activated inside the WebView. Three primary navigation targets are supported:

- **_self**: loads the URL directly inside the main application WebView.

- **_blank**: opens the URL in an InAppBrowser instance, which acts as an embedded browser separate from the main WebView.
- **_system**: delegates the URL to the device's default external browser or application handler.

These navigation modes play a critical role in security because only the main application WebView runs with Cordova's privileged environment and has access to the JavaScript bridge and plugins. URLs opened in the InAppBrowser or the system browser do not (and cannot) access Cordova APIs.

File Loading in Cordova

A major architectural change introduced in Cordova Android 10 (released in 2021) modified the default origin of Cordova applications. Historically, Cordova loaded application assets using a `file://` URL:

`file:///android_asset/www/index.html`

This origin exposed the application to several inconsistencies in the enforcement of the SOP and allowed attacks that relied on loading arbitrary `file://` resources inside the WebView.

Starting from Cordova Android 10, the default origin became:

`https://localhost`

This change significantly improves security and predictability:

- The WebView no longer allows navigation to absolute `file://` paths such as `file:///android_asset/www/...`.
- Only HTTP(S)-based loading is permitted, blocking many previously possible local file access attacks.
- The `https://localhost` origin ensures consistent SOP enforcement across Android versions.

  With the new origin, Cordova internally maps local application files to the `https://localhost` scheme. As a result, local assets can be loaded in two ways:

  - **Relative paths**, such as:

    `js/index.js`

  - **Rooted paths** under the localhost origin, such as:

    `https://localhost/js/index.js`

Both resolve to the same internal asset loader, but they are processed using standard web security rules rather than the legacy `file://` behavior.

Because of this, previously exploitable behaviors—such as loading arbitrary `file://` URLs into iframes or script tags—are effectively blocked in the main WebView. Attempts to load absolute file paths result in navigation failures or blocked requests.

Although the main application, WebView, enforces this new restriction, the InAppBrowser plugin behaves differently. The InAppBrowser operates like a standard browser container with-

out Cordova's special asset loader and therefore retains the ability to load local `file://` paths, for example:

```
file:///android_asset/www/js/index.js
```

or other absolute paths inside the application package.

This distinction is important because:

- The InAppBrowser does not have access to the Cordova JavaScript bridge.
- Loading local files in InAppBrowser does not expose plugin APIs.
- However, attackers may attempt to use it for phishing or misleading UI attacks.

2.2.2 Security Model and Risks

The security of Cordova applications relies heavily on the assumption that the JavaScript code running in the WebView is trusted and originates from a safe source. In practice, however, this assumption is often violated. Many hybrid applications load content from remote servers or use external JavaScript libraries. If an attacker can control or manipulate any of this content, they may be able to execute arbitrary code within the WebView and use the Cordova bridge to perform privileged operations.

This type of vulnerability is particularly dangerous because Cordova effectively grants the WebView access to native capabilities that are normally protected by the Android permission system. In other words, compromising the web layer of a hybrid app can lead to a full compromise of the native layer, a risk rarely present in traditional web applications.

To mitigate these issues, Cordova provides several configuration options and recommendations, such as:

- Defining strict domain whitelists using the `Content-Security-Policy` meta tag.
- Avoiding remote code loading and embedding only trusted local resources.
- Limiting the set of active plugins to the minimum required by the application.
- Keeping the framework and plugins up to date to patch known vulnerabilities.

Despite these measures, empirical studies have shown that many applications fail to follow best practices, often due to a lack of awareness or the perceived complexity of configuring security options correctly. These weaknesses make Cordova-based apps an appealing target for attackers, especially those exploiting injection or privilege escalation vulnerabilities.

2.2.3 Plugin System

In Cordova, each plugin acts as a bridge module between the web layer and a specific native functionality. Plugins can be official (maintained by the Cordova community) or third-party modules created by developers to extend the framework's capabilities. They are typically defined by a JavaScript interface and a corresponding native implementation.

However, this flexibility also introduces security concerns. Plugins often run with elevated privileges and can access sensitive resources such as the file system, network interfaces, or hardware sensors. If a plugin is poorly implemented, outdated, or comes from an untrusted source, it may expose critical vulnerabilities. Moreover, developers frequently include third-

party plugins without conducting thorough security audits, thereby increasing the application's attack surface.

Another source of risk arises from improper configuration of the Cordova `config.xml` file, which defines permissions, whitelisted domains, and plugin usage policies. Overly permissive configurations - for example, allowing remote code execution via the `allow-navigation` or `allow-intent` directives - can enable attackers to inject malicious scripts or exfiltrate user data.

2.3 Ionic Framework

The Ionic framework was introduced in 2013 as a modern and opinionated ecosystem for building hybrid mobile applications. While Ionic is often described as a standalone solution, its early versions relied heavily on Apache Cordova for native functionality. Ionic provided the UI components, tooling, and development workflow, while Cordova supplied the underlying WebView container and the plugin system for accessing native APIs.

Over time, however, the hybrid development landscape evolved, and the limitations of Cordova became more apparent, particularly in terms of maintainability, plugin fragmentation, and reliance on legacy WebView features. In response, the Ionic team introduced *Capacitor*, a new native runtime designed to replace Cordova as the primary bridge between the web layer and the underlying mobile platform. Capacitor now serves as the default native layer for modern Ionic applications.

2.3.1 Architecture and Rendering Model

Regardless of whether Cordova or Capacitor is used as the native runtime, Ionic applications are rendered inside a WebView. The core of an Ionic application comprises HTML templates, TypeScript/JavaScript code, and CSS styles that define the app's interface and logic. These assets are bundled into the native application and served locally to the embedded WebView.

Ionic's UI layer provides a comprehensive set of components that adapt their styling based on the target platform. These components are implemented using modern web frameworks such as Angular, React, or Vue, enabling developers to build complex interfaces with familiar web technologies. The rendering remains web-based, but the user experience aims to emulate native UI patterns.

The difference between Cordova and Capacitor lies not in how the UI is rendered, but in how the web layer communicates with the native layer and how the native container is managed.

2.3.2 Capacitor as the Modern Ionic Runtime

Capacitor represents a new approach to bridging web code with native functionality. Unlike Cordova, which evolved incrementally over more than a decade, Capacitor was designed from the ground up with modern mobile requirements in mind. Its architecture simplifies the interaction between the WebView and native APIs, and introduces several improvements:

- A more consistent and streamlined plugin system.
- First-class support for modern WebView features and platform standards.
- Easier integration with native mobile development tools.

- A persistent file structure that aligns more closely with standard Android and iOS projects.
- APIs designed to reduce reliance on legacy Cordova bridges.

Capacitor applications run their web code inside a WebView, similar to Cordova, but they benefit from a simplified bridging mechanism that allows JavaScript to call native plugins using a standardized and modern interface. Plugins are packaged as regular native modules, and developers can write custom native code more easily compared to Cordova.

The empirical evaluation conducted in this thesis focuses specifically on Ionic applications built on top of Capacitor, as Cordova-related vulnerabilities have already been extensively explored in the previous framework. Moreover, Capacitor is now the default runtime for new Ionic projects and represents the direction in which the ecosystem is evolving.

2.3.3 Security Considerations

Although Capacitor introduces improvements in architecture and plugin management, it does not eliminate the fundamental security challenges inherent in hybrid development. The application still relies on a WebView to execute its web layer, and therefore remains susceptible to threats originating from untrusted or invalidated HTML and JavaScript code.

Security risks may arise from:

- improperly configured WebView settings,
- use of remote or dynamically generated content,
- inclusion of vulnerable third-party JavaScript libraries,
- overly permissive plugin usage or navigation policies,

- misconfigured `Content-Security-Policy` headers.

While Capacitor improves sandboxing and plugin isolation relative to Cordova, the boundary between the web layer and the native layer remains a critical point of exposure. If an attacker gains control of JavaScript executed within the WebView, they may be able to invoke native functionality through the Capacitor bridge, depending on the configuration and installed plugins.

2.4 React Native Framework

React Native, introduced by Facebook in 2015, represents a different paradigm within the broader landscape of cross-platform mobile development. Unlike Cordova and Ionic, which rely on a WebView to render their user interface, React Native enables developers to build mobile applications using JavaScript while rendering actual native components on the device. This approach allows React Native applications to achieve a near-native user experience, both in terms of performance and visual consistency.

React Native builds on the concept of declarative UI programming introduced by the React library for web development. Instead of defining HTML templates that are rendered inside a WebView, developers write React components that describe the desired user interface. At runtime, these components are translated into platform-specific native UI elements such as Android Views or iOS UIViews. This architectural difference places React Native closer to traditional native development than to hybrid WebView-based solutions.

2.4.1 Architecture and Rendering Model

React Native consists of three main layers: the JavaScript layer, the bridge, and the native layer. Application logic is written in JavaScript and executed using the JavaScript engine bundled with the app (Hermes, JSC, or V8, depending on configuration). When UI elements or native functionality are required, the JavaScript code issues commands that are passed through the *React Native bridge*.

The bridge acts as a communication channel between the JavaScript runtime and the underlying platform's native APIs. It serializes messages between the two environments and coordinates the creation, update, or removal of UI components on the screen. Crucially, React Native does not embed a WebView to render its interface; instead, it creates real native widgets (like `<View>` and `<Text>`) controlled programmatically by JavaScript.

Due to its architecture, React Native typically achieves better performance than WebView-based frameworks. Interfaces render more smoothly, animations are more fluid, and applications behave more like fully native apps. However, performance is still limited by the asynchronous nature of the bridge and the overhead involved in serializing messages between JavaScript and the native environment.

2.4.2 Plugin System and Native Modules

React Native exposes native capabilities through *Native Modules*, which are conceptually similar to Cordova and Capacitor plugins, but differ significantly in implementation. Native Modules are written directly in Java (Android), Swift/Objective-C (iOS), or Kotlin, and they provide callable interfaces that the JavaScript code can access.

The React Native ecosystem includes a rich collection of official and community-maintained modules. These cover common functionalities such as camera access, geolocation, sensors, storage, and network operations. Developers may also create custom modules to extend the framework's capabilities beyond the existing library.

While this modularity allows React Native to integrate seamlessly with native APIs, it also introduces potential risks. Poorly maintained or insecure third-party modules can expose sensitive functionality, and their tight coupling with native code means that vulnerabilities may have a broader impact than in purely JavaScript-based environments.

2.4.3 Security Considerations

Although React Native mitigates many of the weaknesses associated with WebView-based hybrid frameworks, it introduces its own security model and associated risks. Since React Native does not rely on a WebView, it is less susceptible to classic injection attacks such as DOM-based script injections, iframe abuses, or manipulations of HTML content.

However, React Native applications still execute untrusted or remote JavaScript code if developers choose to enable features such as over-the-air (OTA) updates via services like CodePush. When external JavaScript is dynamically fetched and executed, the application may become vulnerable to arbitrary code execution, script tampering, and supply-chain attacks. The impact of such an attack is significant because compromising JavaScript execution effectively grants the attacker indirect access to native APIs through the bridge.

React Native also depends heavily on its Native Modules ecosystem. Third-party modules may expose overly permissive interfaces, lack proper security checks, or rely on outdated native

libraries. Vulnerabilities in these modules can allow unauthorized file access, insecure storage of sensitive data, or unintended privilege escalation.

Ultimately, React Native remains subject to the general security models of Android and iOS. Misconfigured permissions, insecure network requests, improper use of HTTPS/TLS, and unsafe storage practices can impact React Native applications in the same way they affect native apps.

2.5 WebView Security Model

The WebView component plays a central role in hybrid mobile applications. It functions as an embedded browser that renders HTML, CSS, and JavaScript inside a native application, enabling developers to reuse web technologies while still distributing their applications through official app stores. In frameworks such as Cordova and Ionic (including their Capacitor-based variants), the WebView is the primary execution environment for the application's UI and a significant portion of its logic.

Although the WebView shares many characteristics with traditional mobile browsers, its security model differs in important ways. Unlike a standalone browser, a WebView operates within the privilege context of the hosting application. This means that vulnerabilities in WebView-executed JavaScript can potentially escalate far beyond the browser sandbox, reaching native APIs through JavaScript bridges exposed by the framework's plugin system. Understanding the security mechanisms of WebView and potential bypasses is crucial for assessing the risks of hybrid mobile applications.

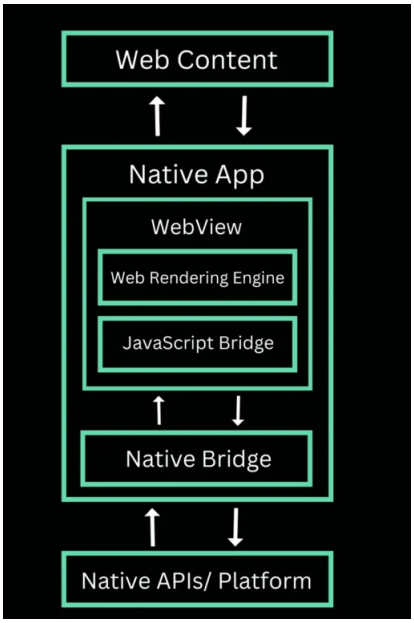

Figure 7: WebView architecture.

2.5.1 Execution Environment and Sandboxing

In a standard mobile browser, web content is confined to a strict sandbox enforced by the browser engine. This sandbox restricts access to the file system, hardware sensors, and other system-level resources. The WebView, however, runs within the sandbox of the hosting Android application and therefore inherits any permissions granted to that application. For example, if the application has permission to access the camera or read external storage, JavaScript running inside the WebView may indirectly gain access to these resources through the plugin system.

This architectural difference means that the security boundary in hybrid applications shifts from a strong, well-defined browser sandbox to a more fragile boundary between JavaScript and

native code. As a result, attacks targeting the WebView environment can have significantly more severe consequences than similar attacks in traditional web applications.

2.5.2 JavaScript Bridges and Native Interaction

A key aspect of the WebView security model in hybrid applications is the presence of JavaScript bridges. These bridges allow JavaScript code to access native functionality through specific interfaces. For example, Cordova and Capacitor utilize their respective plugin systems to provide APIs for various features such as camera access, file operations, geolocation, and device information. Frameworks like Cordova, Capacitor, and, in a different way, React Native, enable this access to native APIs for JavaScript code running within the application. This connection is facilitated through:

- Cordova's `exec` bridge,
- Capacitor's plugin interface,
- React Native's asynchronous message bridge for Native Modules.

From a security perspective, the bridge represents a privileged execution gateway: any JavaScript that gains access to the bridge can potentially invoke sensitive native operations. If an attacker succeeds in injecting malicious JavaScript into the WebView through external scripts, manipulated HTML, or inadequate user input handling, they could execute actions with the same permissions as the application itself.

The presence of the bridge, therefore, transforms many classic web vulnerabilities, such as cross-site scripting (XSS) or DOM injection, into far more damaging attacks that can compromise user data, access device files, or transmit sensitive information to remote servers.

2.5.3 Same-Origin Policy in WebViews

The SOP is one of the most fundamental mechanisms for securing the web. It restricts how documents or scripts from one origin can interact with resources from another origin, preventing unauthorized cross-origin data access.

However, the enforcement of SOP within WebViews differs from that in modern browsers. In many hybrid frameworks, application assets are loaded from local file URLs such as:

```
file:///android_asset/www/index.html
```

Resources loaded from the `file://` scheme are treated inconsistently across different Android versions and WebView implementations. In some configurations, `file://` origins may bypass SOP restrictions, allowing scripts within the application to request arbitrary local files or interact with content not normally accessible to a remote website.

These inconsistencies create opportunities for attackers to exploit injection-based vulnerabilities. For example, a malicious iframe loaded under the same `file://` origin may gain access to local application files or Cordova's File Plugin, depending on the configuration of the WebView.

2.5.4 Content Loading Policies

Security in WebViews also depends on the application's content-loading policies. Hybrid frameworks typically allow developers to specify which external domains can be loaded or

executed within the WebView. In Cordova and Capacitor, this configuration is controlled in the `config.xml` file. Additionally, security headers such as the `Content-Security-Policy` (CSP) can further restrict the execution of inline scripts, external files, or dynamic code.

However, many applications configure these settings improperly. Common mistakes include:

- Allowing all external navigation through wildcard policies (e.g., <`allow-navigation href="*"`>).
- Permitting external JavaScript from untrusted domains.
- Using overly permissive CSP directives such as `script-src *` or `unsafe-inline`.
- Loading remote HTML or templates dynamically without proper sanitization.

Such misconfigurations significantly increase the attack surface of hybrid applications, allowing attackers to inject or manipulate JavaScript content that executes with elevated permissions.

2.5.5 WebView Settings and Configuration

Beyond high-level policies, certain WebView settings can have a direct effect on application security. Some examples include:

- `setJavaScriptEnabled(true)` - required for hybrid apps, but expands attack surface.
- `addJavascriptInterface()` - exposes Java objects to JavaScript; dangerous if used incorrectly.
- `setAllowFileAccess()` and `setAllowUniversalAccessFromFileURLs()` - can permit unauthorized file access.

- `setDomStorageEnabled(true)` - enables Web Storage APIs which may contain sensitive data.

If these settings are not configured carefully, the WebView may allow malicious JavaScript to reach local files, bypass SOP, or call into privileged native methods.

CHAPTER 3

METHODOLOGY

The analysis presented in this thesis required the collection, decompilation, and examination of a large number of real-world hybrid Android applications. Because hybrid applications integrate both web-based and native components, their analysis involves tools and techniques from mobile security, static code analysis, and web security assessment. This chapter describes the methodological approach adopted in this work, covering manual testing on mock applications, dataset acquisition, identification of relevant frameworks, decompilation steps, and the design of the automated script used to detect security weaknesses.

The overarching objective of the methodology is to enable a reproducible and scalable evaluation of the security posture of Cordova, Ionic (Capacitor-based), and React Native applications. By combining publicly available application repositories with automated analysis tools, this approach enables a systematic investigation of framework-specific vulnerabilities and common developer misconfigurations.

3.1 Exploratory Analysis and Manual Testing of Frameworks

Before conducting large-scale automated analysis, it was essential to gain a practical understanding of how each framework behaves in controlled conditions. For this reason, an initial exploratory phase was conducted, during which small mock applications were developed using Visual Studio Code. The frameworks explored were Cordova, Ionic (with its modern Capacitor

runtime), React Native, and Flutter, although the last one was abandoned for further exploration due to its ad hoc programming language (Dart), making it immune to JavaScript code injections. These applications were intentionally minimal, containing only the components necessary to observe the frameworks' configuration mechanisms, execution models, and interactions between JavaScript and native code.

The purpose of this exploratory phase was twofold. First, it provided direct insight into how hybrid and cross-platform frameworks configure their WebView environments, manage permissions, register plugins, and expose native APIs. Second, it allowed the identification of potential vulnerability patterns that could later be formalized into automated detection rules.

Each mock application was built with different combinations of settings in order to test their impact on security. Examples include enabling or disabling file access within the WebView, adjusting navigation whitelists, modifying CSP directives, introducing external scripts, and manipulating the SOP through intentionally crafted file loads or iframe injections. These tests revealed how each framework enforces, relaxes, or bypasses web security guarantees depending on its configuration. To ensure a controlled and verifiable testing environment, a dedicated server hosted at the UIC was employed. This infrastructure served a dual purpose: first, to rigorously simulate the unauthorized extrapolation of sensitive data, and second, to authentically replicate the behavioral patterns of an external third-party entity within the network ecosystem.

The mock applications were initially executed on emulators, but later, a physical Android device was used. Logs, network traffic, and internal framework debug messages were exam-

ined to understand how the frameworks handled navigation requests, plugin initialization, and JavaScript bridge communication.

Insights gained from this exploratory testing phase provided the foundation for the vulnerability classes discussed in Section 3.2. Moreover, the empirical observations collected during this stage directly informed the design of the unified static analysis script described in Section 3.4. By validating the security implications of framework behaviors in controlled environments prior to automation, the methodology ensures that the final detection rules are grounded in real, observable phenomena rather than theoretical assumptions.

3.2 Dataset Collection

To perform a meaningful evaluation of hybrid application security, it was essential to gather a representative and diverse dataset of Android applications built using the targeted frameworks. The dataset used in this thesis was constructed from three primary sources:

- **AndroZoo Library**: A large-scale dataset of Android applications collected from multiple markets, providing APKs along with metadata such as package names, version codes, and cryptographic hashes.
- **APKMirror**: A widely used third-party distribution platform that hosts numerous versions of popular Android applications.
- **APKPure**: Another alternative Android marketplace offering a broad range of applications across different categories and regions.

Applications were selected for download using the official framework website, which includes examples of apps that use the framework, websites that mention the most famous apps built with that specific framework, and random samples from the AndroZoo database, filtering only those in the Google Play Store, obtained through the command:

```
zcat latest.csv.gz | awk -F, 'if ($11 ~ /play\.google\.com/) print '.
```

A filtering step was therefore required to identify which applications were built using Cordova, Ionic, or React Native.

For each downloaded application, associated metadata such as the APK's package name, version, and source was stored to support later analysis.

3.3 Static Analysis Script

A central component of this thesis is the development of a single, unified static analysis script designed to automatically process Android applications and detect framework-specific vulnerabilities. Instead of relying on multiple independent tools, the script integrates all stages of the analysis pipeline, from framework identification to targeted vulnerability scanning. This design choice improves reproducibility, reduces operational complexity, and ensures consistent behavior across different categories of hybrid applications. The script can be found at this GitHub repository: `https://github.com/Civita2107/app_scanner`.

The script operates in multiple phases. First, it inspects the decompiled APK directory and identifies the framework used by the application. This step relies on the same markers described in Section 3.2, such as the presence of Cordova configuration files, Capacitor assets used by modern Ionic applications, or React Native bundle files. Once the framework is determined,

the script automatically invokes the corresponding analysis routine embedded within the same codebase.

Each framework scanner is implemented as an internal module within the script. These modules examine framework-specific assets, configuration files, and architectural patterns to detect misconfigurations or known vulnerability classes. For example, the Cordova module inspects the `config.xml` file, WebView policies, plugin declarations, and JavaScript assets to identify patterns such as external script injection or improper file access restrictions. The Ionic (Capacitor-based) module evaluates the Capacitor configuration and WebView runtime options, while the React Native module analyzes the JavaScript bundle, native modules, and bridge configuration to highlight potential security risks.

By consolidating all logic into a single executable script, the analysis process becomes highly scalable. The script can be executed over large batches of applications with minimal manual intervention, and all results are produced following a uniform structure. This unified approach also simplifies maintenance and reduces the likelihood of inconsistent results arising from different toolchains or version mismatches.

For each analyzed application, the script produces a structured output file containing detected vulnerabilities, framework classification, relevant configuration details, and additional metadata. These results form the basis for the comparative analysis presented in subsequent chapters.

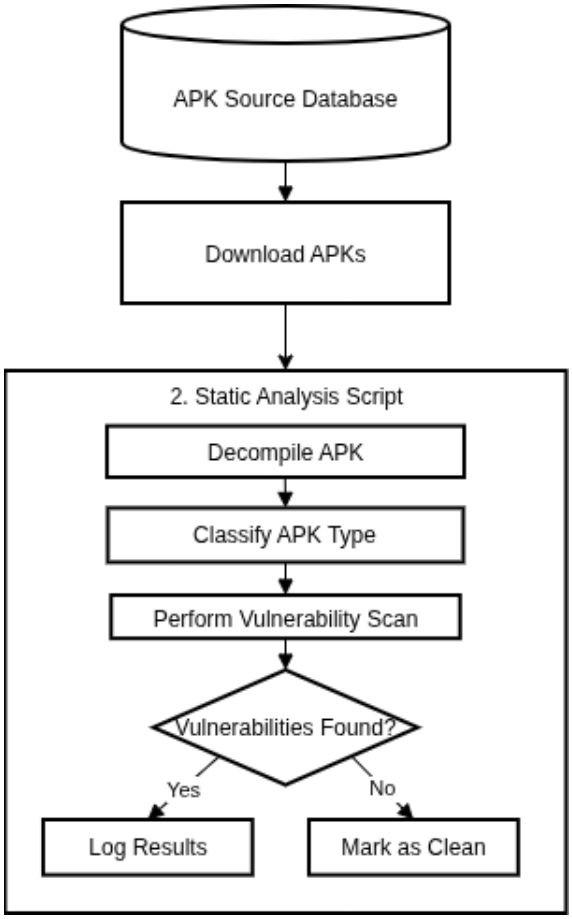

Figure 8: Overview of the methodology pipeline.

3.3.1 Decompilation Pipeline

After identifying the relevant applications, each APK underwent a decompilation process to extract:

- Java/Kotlin source code (via `jadx`),
- JavaScript assets (HTML, JS, CSS),

- WebView configuration files,
- Framework-specific configuration files (e.g., `config.xml`, `capacitor.config.json`),
- Native module implementations (for React Native).

The decompilation pipeline was designed to operate automatically, enabling the processing of dozens or hundreds of applications in sequence. Errors such as corrupted APKs, unsupported formats, or obfuscation issues were logged and handled gracefully.

3.3.2 Framework Identification

Identifying whether an Android application was built with Cordova, Ionic, or React Native required inspecting the application's package contents. After unpacking each APK, the following indicators were used to classify the framework:

- **Cordova**: Presence of `assets/www/cordova.js`, `cordova_plugins.js`, `res/xml/config.xml`, and Cordova-related directory structures.
- **Ionic (Capacitor-based)**: Presence of `assets/capacitor.config.json`, Capacitor Java packages (e.g., `com.getcapacitor`), and web asset folders consistent with Ionic builds.
- **React Native**: Presence of `index.js`, React Native Java modules (e.g., `com.facebook.react`), `ReactNativeHost` classes, and `assets/index.android.bundle`.

Applications were classified based on these characteristic indicators. In cases where multiple frameworks were present, the classification prioritized the dominant runtime (e.g., Capacitor over Cordova when both were detected due to legacy support files).

This classification step ensured that only Android applications built with the targeted frameworks proceeded to the static analysis phases.

3.3.3 Automated Vulnerability Scanning

Once an application has been decompiled and its framework identified, the static analysis script executes the framework-specific scanning routine embedded within the same codebase. This phase implements the detection logic derived from the exploratory analysis (Section 3.1) and the vulnerability taxonomies described in Section 3.2. The scanning routine follows a deterministic sequence of checks designed to minimize false positives while being robust to minor variations in project structure or obfuscation.

Scanning workflow. The scanner proceeds through the following ordered steps for each decompiled APK:

1. **Asset indexing:** Enumerate extracted files and build an index of Java/Kotlin sources, resource manifests, JavaScript bundles, HTML assets, plugin descriptors, and configuration files. This index allows targeted queries without repeated file system traversal.
2. **Configuration parsing:** Parse framework configuration files (e.g., `res/xml/config.xml`, `capacitor.config.json`, AndroidManifest.xml) to extract declared permissions, navigation directives (`allow-navigation`, `allow-intent`, `access-origin`), whitelists, and plugin declarations.
3. **Web asset analysis:** Analyze HTML, CSS and JavaScript assets for indicators of insecure patterns:

- presence of `localStorage` usage and suspicious keys,
- insecure CSP headers or overly permissive `meta` CSP tags,
- dynamic script inclusion patterns (e.g., `document.write`, `eval`, script tags with remote src).

4. **Plugin and native API inspection:** Inspect plugin descriptors and native code for use of sensitive APIs (FileSystem, Camera, Clipboard, Preferences, AsyncStorage bridges). For React Native, search for Native Module registrations and calls to `postMessage`, `injectedJavaScript`, or URL parameter patterns that carry secrets.
5. **Vulnerability rule application:** Apply framework-specific detection rules (see below) that map observed artifacts to one or more vulnerability classes. Each rule comprises a condition (file presence, code pattern, configuration flag) and a confidence score indicating the likelihood of an actual vulnerability.
6. **Heuristic validation and de-duplication:** For matches with medium confidence, attempt secondary validation (e.g., cross-check a declared permission with actual API usage). De-duplicate multiple findings that refer to the same underlying issue to avoid inflating counts.
7. **Result tagging:** Annotate each finding with metadata including affected file paths, line/snippet examples (when available), rule identifier, confidence level, and mitigation hint.

3.3.4 Framework-specific Detection Strategy

The static analysis tool incorporates separate detection strategies for Cordova, Ionic/Capacitor, and React Native.

Each strategy inspects framework-specific files, configuration structures, and runtime patterns to identify signals relevant to hybrid application security.

Rather than applying a single set of checks to all applications, the script activates the appropriate detection routine once the framework has been identified.

These routines examine characteristic markers—such as Cordova configuration directives, Capacitor plugin usage, or React Native WebView integration patterns to extract security-relevant indicators from each application.

3.3.5 Output Structure

The final phase of the static analysis script consists of producing a structured, machine-readable report that summarizes the detected framework, the extracted security-relevant configuration details, and the vulnerability verdicts associated with the analyzed application. Each report is generated in JSON format to ensure interoperability with downstream tools and to facilitate statistical aggregation. The report contains the following top-level fields:

- `apk`: Absolute path to the processed APK file.
- `framework_detection`: A structured object indicating the detected framework (`CORDOVA`, `IONIC`, or `REACT_NATIVE`) together with the justification used for classification (e.g., presence of `config.xml`, Capacitor assets, or React Native bundles).

- `framework`: A shorthand confirmation of the detected framework, provided for convenience.
- `vulnerability_checks`: A set of boolean or categorical flags representing the raw analysis signals observed in the application, such as:
  - whether `localStorage` is used,
  - presence of Cordova File plugin calls,
  - whether the application is debuggable,
  - whether a CSP is present in `index.html`,
  - whether sensitive APIs (e.g., `resolveLocalFileSystemURL`) appear in scripts.
- `security_config`: Extracted configuration indicators including navigation directives (`allow_navigat` `access-origin` usage, presence of wildcards, and other WebView- or plugin-relevant flags.
- `vulnerability_verdicts`: A list of human-readable vulnerability descriptions produced by combining the raw checks with the framework-specific rules. Each entry corresponds to one of the vulnerability classes discussed in Chapter 4.

The following snippet illustrates the structure of a typical report generated for a Cordova application:

```
{
    "apk": "/path/to/app.apk",
    "framework_detection": {
```

```
    "detected": "CORDOVA",
    "reason": "Found key Cordova indicators: config.xml in res/xml,
               cordova.js in assets/www/"
},
"framework": "CORDOVA",
"vulnerability_checks": {
    "internet_permission": true,
    "android_debuggable": false,
    "index_csp_present": false,
    "cordova_plugin_file": true,
    "resolveLocalFileSystemURL_used": true,
    "localStorage_used": true
},
"security_config": {
    "allow_navigation": true,
    "permissive_access_origin": true,
    "wildcard_access_origin": true
},
"vulnerability_verdicts": [
    "External Script Injection accessing Cordova File Plugin API
    (due to: CSP missing and permissive access origin configuration)",
```

```
        "External Script Injection accessing the application HTML files
        (due to: CSP missing and permissive access origin configuration)",
        "Same-Origin Iframe loading of malicious files accessing the
        Cordova File Plugin API (due to: CSP missing and permissive
        access origin configuration)",
        "Same-Origin Iframe loading of malicious files accessing the
        application HTML files (due to: CSP missing and permissive
        access origin configuration)",
        "localStorage data can be exfiltrated (due to: CSP missing
        and permissive access origin configuration)"
    ]
}
```

This output format enables analysis on two complementary levels. First, the low-level configuration signals (e.g., `index_csp_present`, `localStorage_used`) describe exactly which behaviors were observed in the decompiled files. Second, the high-level vulnerability verdicts represent the script's interpretation of these signals, correlating them to determine whether the application exhibits one or more of the vulnerability patterns described in Chapter 4.

This layered output design makes the reports both human-readable for manual validation and structured enough to support large-scale aggregation, cross-framework comparisons, and the generation of summary statistics in later chapters.

3.4 Targeted Dynamic Analysis with Frida

While the primary evaluation in this thesis relies on static analysis, a limited dynamic analysis was conducted to validate the practical exploitability of selected vulnerabilities. The goal of this experiment was not to build a large-scale dynamic testing pipeline, but rather to confirm that specific vulnerability classes identified through static analysis could be leveraged in a real execution environment. To this end, Frida - a widely used dynamic instrumentation framework for Android - was employed to inspect the runtime behavior of Moodle, a Cordova application from the dataset that was found to be vulnerable during the static analysis phase.

3.4.1 Environment Setup

The dynamic analysis required an environment that allowed for arbitrary code instrumentation and WebView inspection. For this purpose, a rooted Android device was used, allowing the installation and execution of the Frida server binary. The setup consisted of the following components:

- **Rooted Android phone**: provided full access to application processes and permitted runtime instrumentation.
- **Frida server**: deployed on the phone and executed with root privileges.
- **Frida client**: installed on the computer to issue commands, attach to processes, and inject instrumentation scripts.
- **USB debugging environment**: enabled communication between the workstation and the device, including access to Chrome remote debugging features.

This setup enabled real-time inspection of the application's WebView and JavaScript contexts.

3.4.2 WebView Instrumentation and Debugging

A custom Frida script was taken from this GitHub repository to enable WebView debugging at runtime. The script attached to the target application's main process and programmatically invoked debugging flags on the Android `WebView` class. This ensured that the WebView instance exposed a remote debugging endpoint, even if the application did not enable it at build time.

Once instrumentation was active, the WebView instance became visible in the Chrome remote debugging interface via `chrome://inspect`. This allowed examination of the following:

- the `index.html` document hosted by the WebView,
- JavaScript execution context,
- loaded scripts and injected resources,
- the state and contents of `localStorage`.

3.4.3 Exploitation of the Vulnerability

Using Chrome DevTools, JavaScript code was executed directly within the application's WebView context through the debugger console. This made it possible to evaluate the feasibility of specific attacks identified during static analysis. Hence, the test verified that:

- `localStorage` keys marked as sensitive in the static analysis were accessible at runtime,
- the WebView allowed script execution originating from DevTools (as a proxy for attacker-controlled injection),

- navigation and resource loading policies permitted by the app configuration matched those identified statically.

During the dynamic analysis, the selected Moodle application, previously identified as vulnerable through static analysis, confirmed the practical exploitability of the detected misconfigurations. In particular, the application's `index.html` file and its active session cookie, stored in the `localStorage`, were accessible from within the WebView's JavaScript context, as exposed through Chrome DevTools. The following images show the code executed in the DevTools console to retrieve the index.html file (the one that fetches the session cookie is analogous), as well as the receipt of the session cookie on the third-party server.

```
> (async function attack() {
      alert('Malicious File Loaded!');

      fetch('http://localhost/index.html')
          .then(response => {
              if (!response.ok) {
                  throw new Error("Network response was not ok: " + response.statusText);
              }
              return response.text();
          })
          .then(data => {
              console.log('File data retrieved successfully!');
              alert('Sensitive data retrieved: ' + data);

              // Send the data to your remote server
              fetch("https://mali92.cs.uic.edu/report", {
                  method: "POST",
                  headers: { "Content-Type": "text/plain" },
                  body: data,
              });
          })
          .catch(err => {
              console.error('Error accessing local file:', err);
              alert('Error accessing local file: ' + err.message);
          });
  })();
```

Figure 9: JavaScript code executed to fetch the index.html file.

```
mali92@mali92:~/fede$ sudo node server.js
[sudo] password for mali92:
HTTPS server running on port 443
HTTP server running on port 80
Report received
{"preferences":{},"localStorage":{"__advancedHttpCookieStore__":"{\"moodle.polito.it\":{\"/\":{\"MoodleSession\":{\"key\":\"MoodleSession\",\"value\":\"qm027eh5m4uqe75m7r50070red\",\"domain\":\"moodle.polito.it\"
,\"path\":\"/\",\"secure\":true,\"extensions\":[\"SameSite=None\"],\"hostOnly\":true,\"creation\":\"2025-10-24T17:29:58.372Z\",\"lastAccessed\":\"2025-11-16T19:20:25.653Z\"}}}}"},"sessionStorage":{},"fingerprint"
:{"userAgent":"Mozilla/5.0 (Linux; Android 16; Pixel 8a Build/BP3A.250905.014; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/141.0.7390.124 Mobile Safari/537.36 MoodleMobile 5.0.0 (50003)","platfo
rm":"Linux aarch64","language":"en-US","cookieEnabled":true,"onLine":true,"screen":{"width":412,"height":915,"colorDepth":24},"timezone":"America/Chicago"},"fileContent":null,"url":"http://localhost/main/home/das
hboard","timestamp":"2025-11-16T19:44:21.243Z"}
```

Figure 10: Retrieval of the session cookie on the server.

A key contributing factor was the application's CSP configuration, which specified a wildcard `*` as the allowed source for network requests. This configuration enabled scripts running inside the WebView to interact with arbitrary remote endpoints, including third-party servers unrelated to the Moodle infrastructure. Because the WebView permitted outbound requests to any origin, it became feasible for injected JavaScript to issue an HTTP `POST` request to a remote server and transmit locally accessible data. As a result, both the retrieved `index.html` file and the session cookie stored within the WebView environment could be programmatically forwarded to an external domain.

This validation step demonstrated that the issues detected by the static analysis—namely, the permissive CSP and unrestricted navigation rules were not merely theoretical risks; they were actual vulnerabilities. Instead, they directly enabled the exfiltration of sensitive application data during real execution. This confirms the relevance of the vulnerability classes defined earlier in the methodology and highlights the security impact of overly permissive WebView configurations in hybrid Android applications.

CHAPTER 4

EXPERIMENTAL FINDINGS

The exploratory testing described in Section 3.1 enabled the identification of security-relevant behaviors within hybrid frameworks. Building on these observations, this section formalizes the vulnerability classes that were identified as characteristic of each framework. These classes served as the foundation for the automated detection logic in the static analysis script.

Because the security architecture varies significantly between Cordova, Ionic (Capacitor-based), and React Native, each framework exhibits different patterns of misconfiguration, misuse, or structural weaknesses. For example, WebView-based frameworks such as Cordova and Ionic expose a large surface area for script injection and SOP violations, whereas React Native's risks are more closely linked to unsafe JavaScript bundle execution, untrusted OTA updates, or insecure native modules.

The subsections that follow define these vulnerability classes individually, explain their significance within each framework, and describe how they guided the implementation of the scanning modules in Section 3.4. This structure ensures that the analysis pipeline is grounded in an explicit and framework-aware threat model.

4.1 Cordova Vulnerabilities

Cordova applications rely on a WebView that renders local HTML and JavaScript files and exposes native capabilities through a privileged JavaScript bridge. Because the platform is historically designed around `file://` origins, permissive navigation rules, and globally accessible plugin interfaces, Cordova applications exhibit a distinctive set of security weaknesses related to script injection, origin relaxation, and unrestricted plugin exposure. During the exploratory analysis, six vulnerability classes were identified. These represent the characteristic risks that arise when Cordova applications are configured without strict WebView hardening or robust Content Security Policies.

External Script Accessing the LocalStorage. Cordova applications frequently store application state, session identifiers, and user-related information using the WebView's `localStorage`. Any JavaScript executed inside the WebView inherits access to these values. When developers omit a restrictive Content Security Policy or allow untrusted navigation, injected scripts can read and exfiltrate sensitive data. Although navigation to a new WebView instance applies a different CSP, the `localStorage` of the original view remains exposed to any script running within that original context.

Same-Origin Iframe Accessing the LocalStorage. Because Cordova relies on `file://` origins, Same-Origin Policy enforcement is inconsistent across Android versions. If an application loads content inside an iframe that resolves to the same origin—either through permissive `allow-navigation` rules or unvalidated remote content—an attacker-controlled iframe can read

or modify `localStorage`. This vulnerability is amplified by Cordova's tendency to treat multiple `file://` locations as origin-equivalent.

External Script Accessing the File Plugin API. When a Cordova WebView permits external script execution—due to missing CSP directives, permissive `access-origin` settings, or uncontrolled script sources—malicious JavaScript gains access to the full Cordova plugin bridge. This includes the File plugin, which exposes internal storage directories such as the application cache, persistent files, or user-generated data. Such access represents a severe escalation beyond typical web-based code injection.

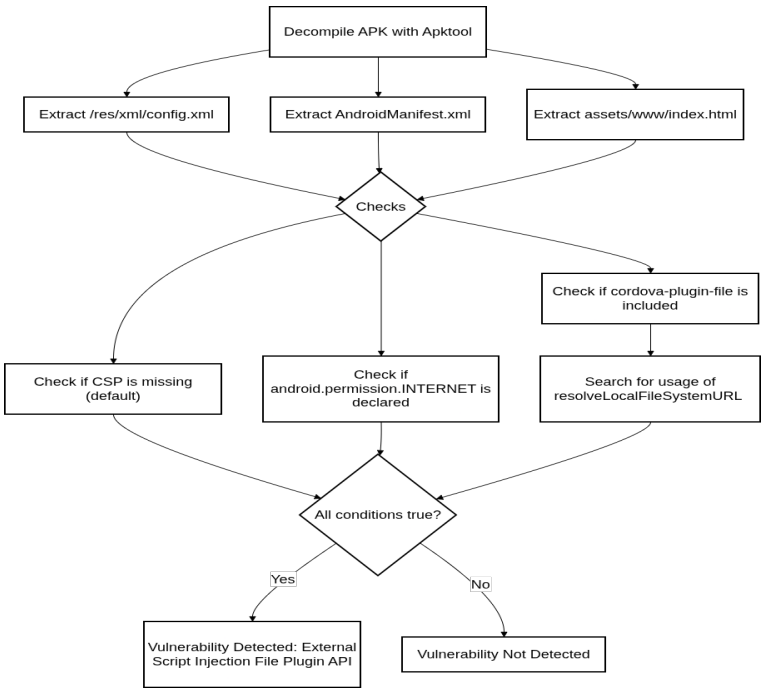

Figure 11: Cordova File Plugin API vulnerability flow.

External Script Accessing the Application HTML Files. In addition to plugin access, injected scripts can retrieve the application's internal HTML and JavaScript files. If the WebView allows `file://` access or universal access from file URLs, attackers can read assets such as `index.html`, bundled JavaScript files, or configuration artifacts. These resources often contain embedded secrets, API endpoints, or logic that facilitate further exploitation.

Same-Origin Iframe Accessing the File Plugin API. Cordova applications that allow broad navigation patterns or relaxed file-access settings may load attacker-controlled HTML inside a same-origin iframe. In this scenario, the iframe inherits the full Cordova JavaScript bridge, including plugin interfaces. As a result, malicious iframe content gains the ability to enumerate files, read application storage, or invoke privileged APIs.

Same-Origin Iframe Accessing the Application HTML Files. Similarly, a same-origin iframe may directly access internal HTML resources if the WebView applies relaxed origin rules. This enables an attacker-controlled iframe to read bundled assets or reconstruct application logic. These behaviors are particularly prominent in legacy Android versions where `file://` equivalence is overly permissive.

These six vulnerability classes form the basis of the Cordova-specific findings from the exploratory phase and directly inform the automated static detection logic described in Section 3.3. Their prevalence highlights the substantial risk posed by permissive WebView configurations, broad navigation allowances, and global plugin exposure within Cordova applications.

A summary table of all Cordova vulnerability tests is provided in Subsection 5.2.1.

4.1.1 Attack Scenario: Same-Origin File Exfiltrating FileSystem.

During the exploratory analysis, an attack on the Cordova File System was conducted on the mock application, resulting in the extrapolation of a file created through the Cordova File Plugin.

In this configuration, any HTML file stored inside the application's asset structure (e.g. `assets/` or `www/`) is considered part of the same origin (`https://localhost`). If the application allows users to download files or store content within these directories, a malicious HTML file may later be rendered in an iframe inside the WebView while still inheriting the trusted application origin.

Once loaded, the attacker-controlled file executes JavaScript with full same-origin privileges. As a result, it gains access to the Cordova JavaScript bridge and can invoke privileged plugins such as the File API. This enables the injected file to:

- read files created by the application in directories like `dataDirectory`,
- load stored files via `resolveLocalFileSystemURL()`,
- exfiltrate the retrieved data to a remote server using `fetch()`.

In the demonstrated attack (Figure 12), the malicious file, once rendered under `https://localhost`, reads a stored file such as `sensitive_data.txt` and transmits its content to an attacker-controlled endpoint. Because the file was executed in a fully trusted origin, no WebView restrictions blocked its access to the Cordova APIs.

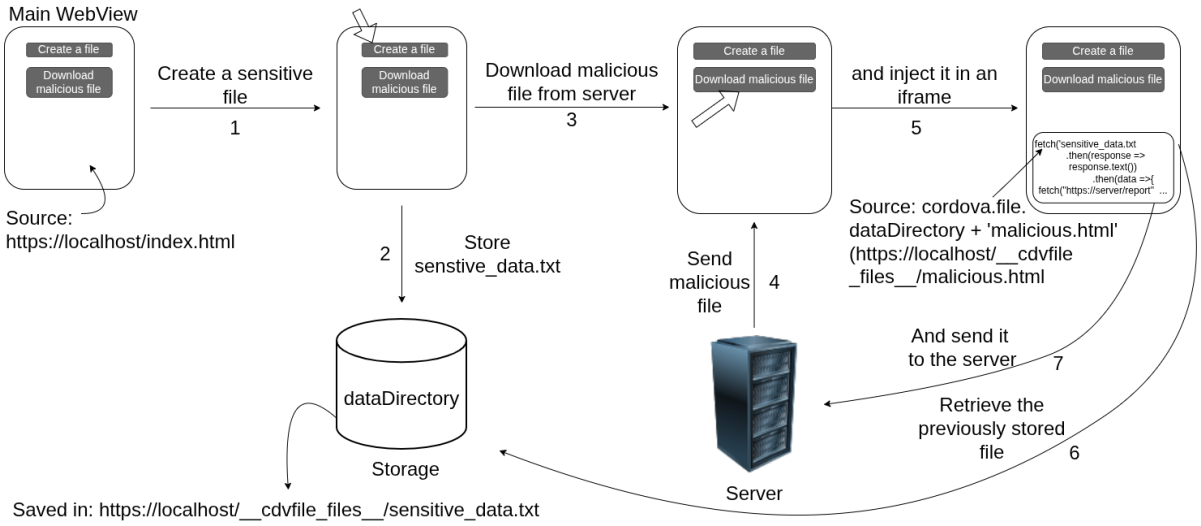

Figure 12: Cordova same-origin file injection enabling file exfiltration.

This vulnerability highlights the risks of allowing untrusted or user-supplied files to be stored in locations that inherit the `https://localhost` origin. When combined with the globally exposed Cordova plugin bridge, any such file becomes a vehicle for full privilege escalation within the WebView environment.

4.1.2 Automated Detection Logic for Cordova

The static analysis script implements a dedicated detection routine tailored to the architectural characteristics of Cordova applications. Once an APK is identified as Cordova-based, the corresponding module inspects a series of configuration files, runtime parameters, and WebView-related assets that reflect the vulnerability patterns validated during the exploratory analysis.

The module first examines the `config.xml` file, which defines navigation rules, origin permissions, and Content Security Policies. Permissive `allow-navigation` entries, wildcard

`access-origin` directives, or missing CSP definitions are recorded as indicators of weakened origin isolation. The detector then inspects the `assets/www` directory to identify the presence of `localStorage` usage patterns, external script references, or inline JavaScript that could be abused through injection.

Furthermore, the module analyzes the WebView configuration by scanning for unsafe Android flags such as `addJavascriptInterface` or `allowFileAccessFromFileURLs`, which may expose sensitive plugin interfaces. It also checks whether the Cordova File plugin is referenced and whether its API calls appear in the decompiled JavaScript. These indicators collectively allow the module to associate the application with the Cordova vulnerability classes described in Section 4.1.

The output of this detection routine provides a high-level summary of the application's security posture, linking configuration signals to the vulnerability classes observed in the manual experiments.

4.2 Ionic Vulnerabilities

Modern versions of Ionic rely on the Capacitor runtime, which replaces the older Cordova backend with a more modular and streamlined architecture. Although Capacitor applications still depend on a WebView, their plugin model, `https://localhost` origin, and permission handling differ substantially from traditional Cordova apps. The exploratory analysis conducted in controlled mock applications revealed seven recurrent vulnerability classes arising from insecure WebView behavior, plugin exposure, and insufficient origin restrictions.

External Script Accessing the LocalStorage. Capacitor does not modify the semantics of the Web Storage API, and all `localStorage` values remain directly accessible to any JavaScript executing inside the WebView. If an attacker manages to inject arbitrary JavaScript or load untrusted same-origin content, these values can be extracted and exfiltrated. The risk becomes more severe when developers store sensitive tokens or user identifiers in `localStorage`.

Capacitor Preferences Access. The `@capacitor/preferences` plugin provides persistent storage via a native-backed key-value store. However, this API is exposed to the WebView without additional filtering. Injected JavaScript, if it gains access to the Capacitor bridge, can read or modify application preferences, potentially altering authentication state or leaking sensitive configuration values.

Capacitor FileSystem Access. The Capacitor FileSystem plugin allows direct file operations on the device. In the presence of script injection, malicious JavaScript may invoke this API to read internal application files or user-generated data. Access to sensitive files represents a significant escalation, extending the impact beyond typical web-based vulnerabilities.

Same-Origin Iframe Accessing the LocalStorage. Ionic applications are served from the `https://localhost` origin. If an untrusted iframe is loaded under the same origin (due to misconfigured routing or overly permissive navigation rules), it inherits the ability to read and modify the same `localStorage` namespace. This enables attacker-controlled documents to gain access to persistent application data.

Same-Origin Iframe Accessing the Capacitor Preferences. Because Capacitor exposes plugin functionality through a global JavaScript interface, same-origin iframes also gain

access to the Preferences API. An attacker-controlled iframe, therefore, inherits the ability to read and modify native-backed persistent state, making this vulnerability particularly impactful.

Clipboard Access Vulnerability. The Capacitor Clipboard plugin allows reading and writing of the system clipboard. If arbitrary JavaScript executes within the WebView, it can retrieve or replace clipboard contents with attacker-controlled data. Depending on user behavior, this may expose sensitive items such as passwords or two-factor authentication codes.

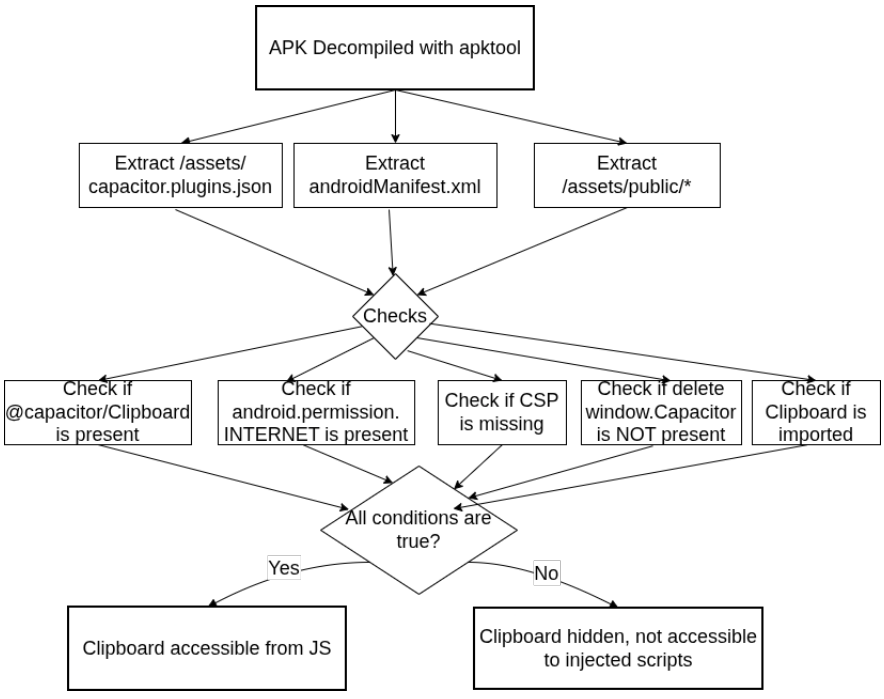

Figure 13: Ionic Clipboard vulnerability flow diagram.

Camera Access Vulnerability. The Capacitor Camera plugin allows web-exposed JavaScript to capture images using the device camera. After the user grants the runtime permission, injected scripts may programmatically trigger camera access, potentially capturing images without the user's legitimate intent. This constitutes a direct privacy risk driven by WebView injection vectors.

These seven vulnerability classes form the basis of the automated detection rules implemented in the static analysis script. They capture the principal security risks observed during manual exploration of Ionic and Capacitor applications, highlighting how plugin exposure and origin misconfiguration can significantly expand the attack surface.

A summary table of the vulnerabilities tested in Ionic applications is provided in Subsection 5.2.2.

4.2.1 Attack Scenario: Capacitor FileSystem Exploitation

During manual testing, an attack scenario was validated on a mock Ionic/Capacitor application to illustrate the practical impact of insecure WebView execution and unrestricted plugin exposure. The scenario highlights the Capacitor FileSystem plugin and illustrates how an injected script, once executed within the WebView, can access and exfiltrate files stored in the application's private directories.

The attack proceeds in three phases. First, the user interacts with a benign application interface that stores files locally through the FileSystem API. Second, the attacker succeeds in injecting arbitrary JavaScript into the WebView, for example, through an untrusted navigation flow or an iframe-based injection vector. Once running, the injected code invokes the FileSys-

tem plugin with the same privileges as the legitimate application, retrieving files stored under the `Directory.Documents` namespace. Finally, the retrieved data is transmitted to a remote attacker-controlled server.

This process is depicted in Figure 14, which illustrates how exposing Capacitor plugins to arbitrary JavaScript enables direct access to native-backed storage and allows for the silent extraction of user or application-generated files.

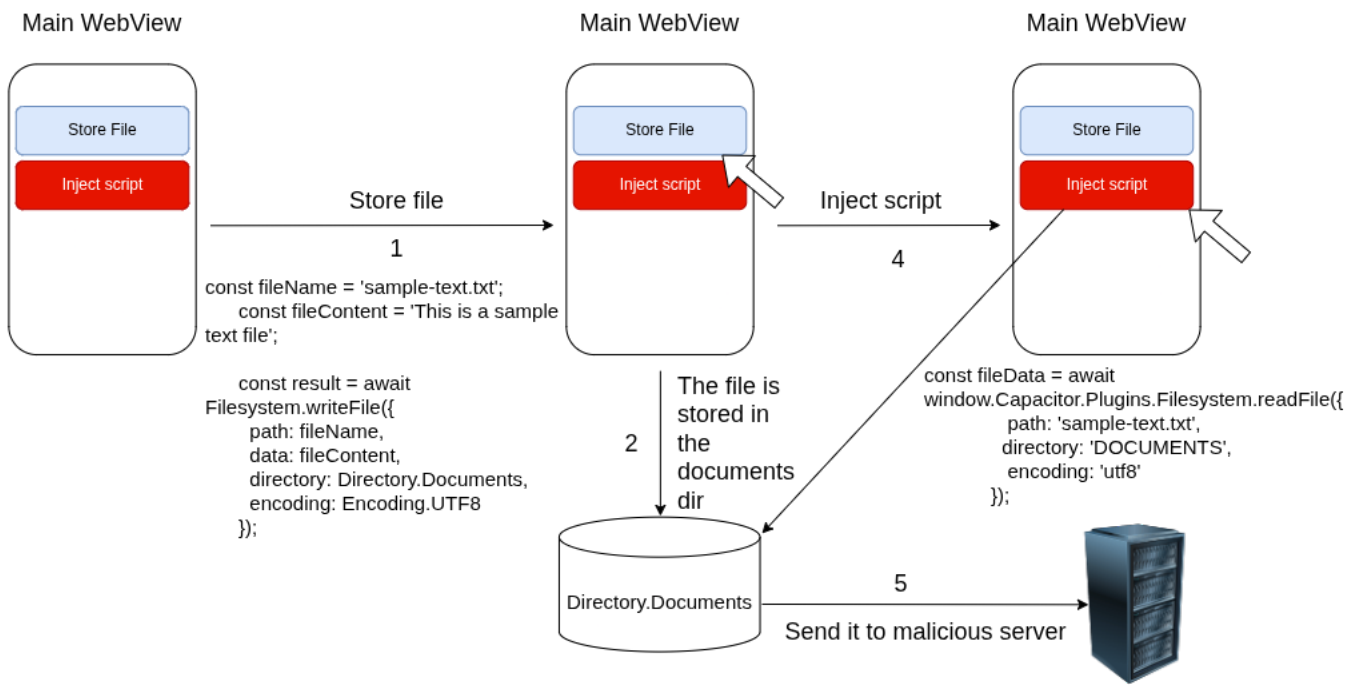

Figure 14: Capacitor FileSystem Exploitation Attack Flow.

This attack confirms that plugin-level vulnerabilities in Ionic are not merely theoretical; they are real and can be exploited. When WebView injection is possible, exposing native APIs such as FileSystem directly enables unauthorized access to local data. This reinforces the need for strict navigation controls, CSP enforcement, and careful limitation of plugin surface available to WebView-executed JavaScript.

4.2.2 Automated Detection Logic for Ionic/Capacitor

For Ionic applications, the detection logic focuses on the Capacitor runtime, which exposes native functionality through globally accessible JavaScript interfaces. Once an application is classified as Ionic/Capacitor, the module inspects the `capacitor.config.json` file, plugin declarations, and the generated WebView assets to extract security-relevant indicators.

The detector searches for evidence of Capacitor plugin usage, including the `@capacitor/preferences`, `Filesystem`, `Clipboard`, and `Camera` APIs. References to these plugins within the JavaScript bundle reveal which privileged operations may be accessible if untrusted code executes inside the WebView. The module also identifies application reliance on `localStorage` or `sessionStorage`, which may expose persistent data in the presence of injection.

To capture navigation-related risks, the module scans for same-origin iframe inclusion, relative asset loading patterns, and any conditions that may unintentionally expose the application to same-origin iframe abuse. These indicators correspond to the Ionic vulnerability classes identified in Section 4.2, including Preference manipulation, Filesystem access, clipboard extraction, and plugin misuse.

By correlating plugin exposure with storage usage and origin-related behaviors, the Ionic/Capacitor detection routine provides a structured assessment of how closely an application's configuration resembles the vulnerable patterns observed in the controlled mock applications.

4.3 React Native Vulnerabilities

Among the three frameworks examined, React Native exhibited the smallest attack surface and the lowest number of detectable vulnerabilities. This advantage primarily stems from its architecture: React Native does not rely on a WebView to render its interface, uses a strongly controlled JavaScript–native bridge, and executes JavaScript logic in an isolated runtime environment (JavaScriptCore or Hermes). Web content is introduced into the application only when the developer explicitly embeds a WebView component. Consequently, many of the WebView-driven weaknesses affecting Cordova and Ionic, like direct access to native interfaces or file system exposure, are not applicable to React Native by default.

However, the exploratory analysis identified a notable vulnerability class originating not from the framework design, but from insecure developer practices that intentionally bridge sensitive native state into a WebView. The unified vulnerability mechanism described below represents the most significant security risk observed in real-world React Native applications.

AsyncStorage Access Vulnerability. Applications frequently store authentication tokens, user identifiers, and other security-sensitive data using `AsyncStorage` or secure alternatives such as `SecureStore`. As long as these values remain within the React Native layer, they are not accessible to WebView-rendered content. The vulnerability arises when developers manually inject stored values into the WebView environment using calls such as:

- `postMessage()` to forward values into the DOM context,
- `injectedJavaScript` to write secrets directly into the page,
- URL parameters appended to the requested resource (e.g., `https://example.com/?token=...`).

Once forwarded into the WebView, these values become subject to the standard browser security constraints rather than React Native's isolation guarantees. If the WebView is configured with permissive settings such as:

- `originWhitelist = ['*']` (arbitrary navigation permitted), or
- `javaScriptEnabled = true` with untrusted content loadable,

then any malicious or compromised page rendered within the component can execute arbitrary JavaScript and read the injected data. The attacker can subsequently exfiltrate the information by issuing simple `fetch()` or XHR requests to a remote server, leveraging the WebView's full browser-like capabilities.

This unified vulnerability scenario represents the primary security risk observed in React Native applications during this research. It highlights that React Native's strong security posture can be undermined not by the framework itself, but by developer decisions that bridge protected application state into a permissively configured WebView environment. Proper origin restrictions, removal of unnecessary WebView integrations, and avoiding the injection of sensitive values into WebViews are essential to maintaining React Native's security guarantees.

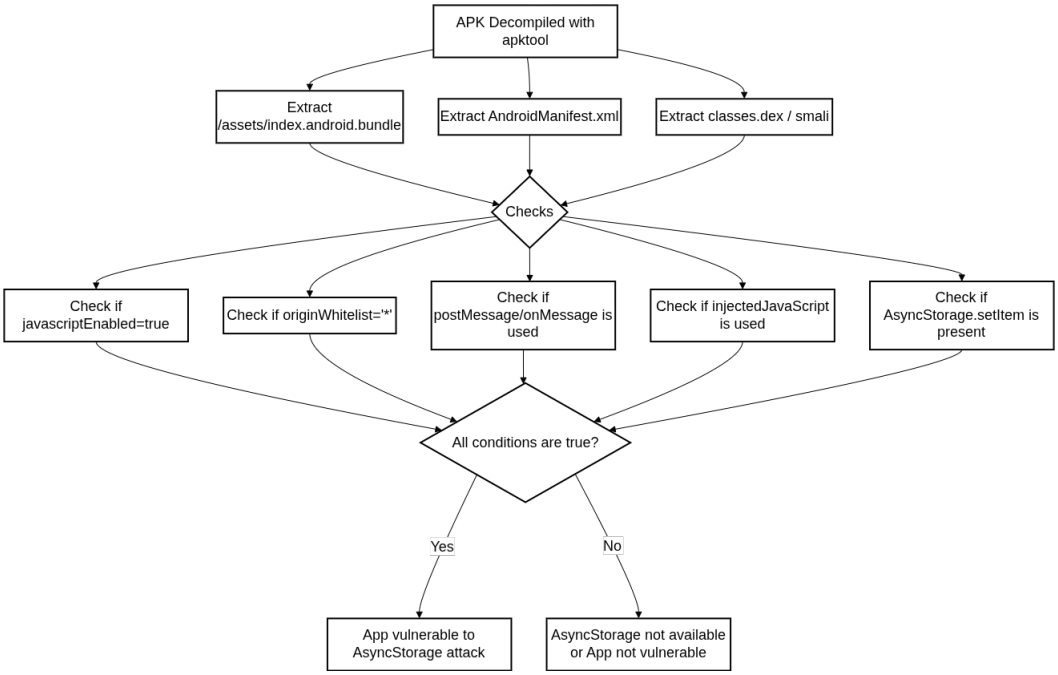

Figure 15: AsyncStorage Vulnerability Diagram Flow.

4.3.1 Attack Scenario: AsyncStorage Exfiltration

Although React Native applications generally exhibit a more secure architecture due to the isolation between the JavaScript execution environment and the native layer, a high-impact vulnerability can occur when developers intentionally bridge sensitive application state into a WebView.

During the exploratory analysis, a realistic attack scenario was demonstrated in which data stored using `AsyncStorage` was exfiltrated through a malicious page rendered inside the WebView. Sensitive data stored with `AsyncStorage` (e.g. tokens, usernames, identifiers) persists

across application sessions and is not automatically protected from disclosure once injected into the WebView context.

The vulnerability does not originate from `AsyncStorage` itself, but from the developer's bridging logic. The WebView provides a two-way messaging channel via `postMessage()`, which allows the native side to send arbitrary values into the web content. For example:

```
webViewRef.current.postMessage(JSON.stringify({ username }));
```

If a developer chooses to forward stored values into the WebView, any JavaScript running inside that page, whether it is legitimate or malicious, gains access to those values and can then exfiltrate them. The WebView context is therefore the only environment at risk: React Native views and other components remain isolated from such manipulation.

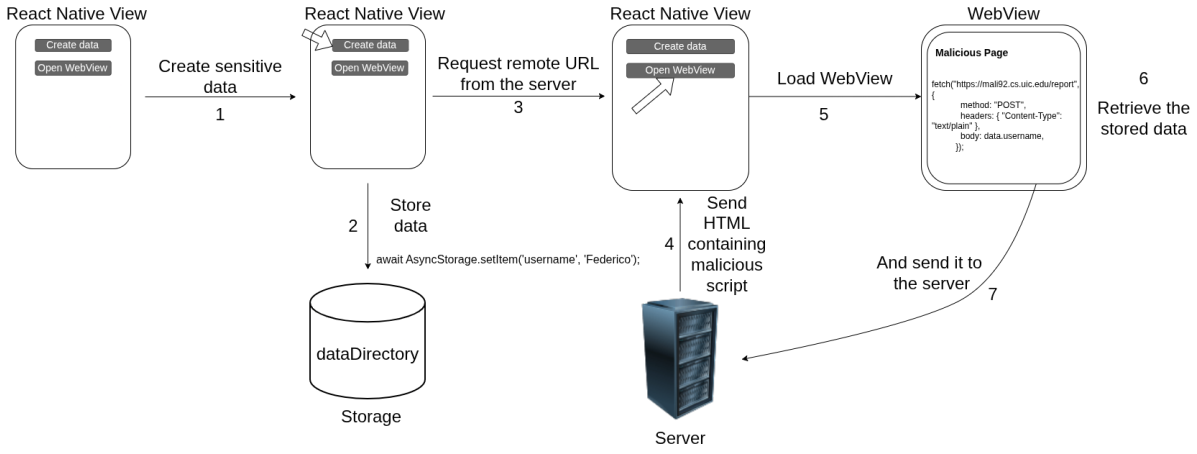

Figure 16: AsyncStorage Exfiltration Attack Flow.

Importantly, this attack (depicted in Figure 16) is only feasible when the developer explicitly bridges sensitive values into the WebView. If the WebView never receives such information, it cannot expose it. However, when bridging is present, any storage mechanism - `AsyncStorage`, `SecureStore`, SQLite databases, native files, or even cloud-retrieved values - becomes vulnerable as soon as the data is passed into the WebView.

This scenario highlights that React Native's strong default isolation can be undermined by insecure developer practices. While the framework does not inherently expose storage to web content, unsafe data bridging reintroduces WebView-based attack vectors similar to those of hybrid frameworks like Cordova and Ionic.

4.3.2 Automated Detection Logic for React Native

The React Native module reflects the more constrained attack surface of this framework. Because React Native does not inherently rely on a WebView, the detection logic focuses on optional components and specific developer choices that may introduce security risks.

The detector first searches the JavaScript bundle (typically `index.android.bundle`) for uses of `AsyncStorage` or `SecureStore`, which represent common storage locations for application secrets. It then examines whether these values are ever forwarded into a WebView environment, for example, through `postMessage`, `injectedJavaScript`, or dynamic URL construction. Such patterns may enable the attack scenario described in Section 4.3, where sensitive tokens can be exfiltrated once introduced into the WebView context.

If the application embeds a WebView component, the module inspects its configuration for permissive settings such as `originWhitelist=['*']` or `mixedContentMode="always"`, as well

as potentially dangerous native bindings created through `addJavascriptInterface`. These checks identify cases where the application inadvertently exposes privileged functionality to injected scripts.

Overall, the React Native detection routine links WebView configuration, storage handling, and message-passing patterns to the vulnerability class validated during the manual experiments, offering a precise view of where React Native applications deviate from the secure defaults established by the framework.

4.4 Dynamic Exploitation Case Study

To complement the static analysis results and validate the practical impact of the detected vulnerabilities, a dynamic exploitation test was performed on a real-world hybrid application. Among the applications identified in the dataset, the Moodle Android client exhibited several high-risk indicators during static analysis, including a missing Content Security Policy (CSP) and permissive navigation directives. Its architecture, based on Cordova, made it an appropriate candidate for demonstrating the runtime implications of these misconfigurations.

During execution, the application's WebView allowed arbitrary JavaScript evaluation, enabling direct inspection of the internal WebView context. Within this environment, both the `index.html` source file and the authenticated session cookie stored in `localStorage` were accessible. Figure 17 and Figure 18 illustrate the retrieval of these resources through JavaScript executed inside the WebView.

```
> (async function attack() {
      alert('Malicious File Loaded!');

      fetch('http://localhost/index.html')
          .then(response => {
              if (!response.ok) {
                  throw new Error("Network response was not ok: " + response.statusText);
              }
              return response.text();
          })
          .then(data => {
              console.log('File data retrieved successfully!');
              alert('Sensitive data retrieved: ' + data);

              // Send the data to your remote server
              fetch("https://mali92.cs.uic.edu/report", {
                  method: "POST",
                  headers: { "Content-Type": "text/plain" },
                  body: data,
              });
          })
          .catch(err => {
              console.error('Error accessing local file:', err);
              alert('Error accessing local file: ' + err.message);
          });
  })();
```

Figure 17: JavaScript executed in the WebView context to retrieve the `index.html` file.


```
mali92@mali92:~/fede$ sudo node server.js
[sudo] password for mali92:
HTTPS server running on port 443
HTTP server running on port 80
Report received
{"preferences":{},"localStorage":{"__advancedHttpCookieStore__":"{\"moodle.polito.it\":{\"/\":{\"MoodleSession\":{\"key\":\"MoodleSession\",\"value\":\"qm027eh5m4uqe75m7r50070red\",\"domain\":\"moodle.polito.it\"
,\"path\":\"/\",\"secure\":true,\"extensions\":[\"SameSite=None\"],\"hostOnly\":true,\"creation\":\"2025-10-24T17:29:58.372Z\",\"lastAccessed\":\"2025-11-16T19:20:25.653Z\"}}}}"},"sessionStorage":{},"fingerprint"
:{"userAgent":"Mozilla/5.0 (Linux; Android 16; Pixel 8a Build/BP3A.250905.014; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/141.0.7390.124 Mobile Safari/537.36 MoodleMobile 5.0.0 (50003)","platfo
rm":"Linux aarch64","language":"en-US","cookieEnabled":true,"onLine":true,"screen":{"width":412,"height":915,"colorDepth":24},"timezone":"America/Chicago"},"fileContent":null,"url":"http://localhost/main/home/das
hboard","timestamp":"2025-11-16T19:44:21.243Z"}
```


Figure 18: Exfiltration of the session cookie to a third-party server.

A central enabling factor for this attack was the application's highly permissive CSP configuration, which defined the wildcard * as an allowed source for network requests. Combined with equally permissive navigation and access-origin rules, this configuration allowed JavaScript running inside the WebView to perform unrestricted outbound HTTP requests. As a result, once sensitive data were accessible inside the WebView, exfiltration to an external domain became trivial.

The test confirmed that the issues flagged by the static analysis were not hypothetical. The combination of a missing CSP, permissive origin policies, and unrestricted file access allowed the extraction of both application resources and authentication material. This demonstrates how insecure WebView configurations can escalate into full data compromise in real-world hybrid applications, reinforcing the importance of the vulnerability classes defined in the preceding sections.

CHAPTER 5

EVALUATION AND RESULTS

5.1 Introduction

This chapter presents the results of the large-scale security evaluation performed on Android applications developed using Cordova, Ionic (Capacitor-based), and React Native. Building on the methodology outlined in Chapter 3, the analysis combines three complementary sources of evidence: (i) framework-specific vulnerability detection carried out on mock applications, (ii) a review of selected high-profile or representative applications identified as vulnerable, and (iii) a brute-force scanning campaign over a substantial subset of the AndroZoo repository.

5.2 Frameworks Results

This section presents the empirical results obtained from the security evaluation of the three hybrid and cross-platform frameworks analyzed in this thesis: Cordova, Ionic (Capacitor-based), and React Native. Each subsection summarizes the outcomes of the controlled attack tests conducted on representative applications developed with the corresponding framework. The results highlight how architectural design choices, default WebView configurations, and plugin exposure models influence the types of attacks that are feasible in each environment. By comparing the behavior of these frameworks under equivalent attack scenarios, this section provides a clear view of their relative strengths, recurring misconfigurations, and common

sources of risk. Detailed findings for Cordova, Ionic, and React Native are presented in the following subsections.

5.2.1 Cordova Results

The analysis of Cordova applications revealed a consistent pattern of vulnerabilities stemming primarily from permissive WebView configurations, missing or weak Content Security Policies, and the broad exposure of Cordova's File Plugin API within the JavaScript context. Table II summarizes the results of the systematic vulnerability tests conducted on representative Cordova applications.

Overall, the evaluation shows that Cordova WebViews remain highly susceptible to external script injection when no restrictive CSP is enforced. Such injected scripts are not only capable of reading application-owned files through the File Plugin API but also accessing local HTML resources such as `index.html` and bundled JavaScript files. These results confirm that inadequate isolation between WebView content and native plugin interfaces constitutes a recurring security issue in Cordova-based applications.

The tests also demonstrate that same-origin iframes inherit the full privileges of the main WebView. When the iframe loads a resource from the application's origin, either through a relative path or the `https://localhost` scheme used by modern Cordova versions, it can invoke native APIs and read application HTML files just as the main WebView can. However, when the iframe loads remote content, the SOP effectively prevents access to local files and plugin interfaces, blocking several attack paths.

Further testing showed that CORS protections are correctly applied: neither local HTML files nor iframe-loaded documents were able to bypass cross-origin restrictions when interacting with remote APIs. Similarly, mixed-content restrictions blocked the loading of cleartext HTTP resources, unless explicitly overridden via the `usesCleartextTraffic` flag in the application manifest.

Finally, localStorage proved accessible to both injected scripts and same-origin iframes, confirming that sensitive data stored in the WebView context remains exposed when the application lacks a strong CSP and employs permissive navigation or plugin access rules.

Taken together, these findings highlight Cordova's continued reliance on secure developer configuration. In the absence of strict CSPs, restricted navigation rules, and controlled plugin exposure, Cordova applications are prone to a range of WebView-driven attacks that can lead to file disclosure, script injection, and unauthorized access to locally stored data.

TABLE II: VULNERABILITIES TESTED IN CORDOVA.

Vulnerability	**Context**	**Target**	**Result**	**Notes**
External script accessing created file	WebView	File Created in the Application	✔ Successful	Can access files created through the File Plugin (saved in dataDirectory, cacheDirectory, etc.), also cross-directory
External script accessing HTML files	WebView	Application HTML files	✔ Successful	Can fetch local files like **index.html** and **index.js** through relative path (../index.html) or absolute path (https://localhost/index.html)
Cross-origin iframe accessing created file	Iframe (file from remote server)	File Created in the Application	**X** Blocked	**SOP** blocks access to Cordova File Plugin API
Cross-origin iframe accessing created file	Iframe (file from remote server)	Application HTML files	**X** Blocked	**SOP** blocks access to local files
Same-origin iframe accessing created file	Iframe (file from app origin)	File Created in the Application	✔ Successful	Since the file is loaded from within the app, it can access Cordova File Plugin API
Same-origin iframe accessing HTML files	Iframe (file from app origin)	Application HTML files	✔ Successful	Since the file is loaded from within the app, it can access local files
CORS fetching from local file	Local HTML file	External API with CORS protection	**X** Blocked	**CORS** is respected
CORS Fetching from Iframe	Malicious file loaded in iframe	External API with CORS protection	**X** Blocked	**CORS** is respected
Cleartext HTTP Loading	WebView, iframe, InAppBrowser	External website	**X** Blocked	Mixed content is blocked by default, but it can be allowed by setting **usesCleartextTraffic=true** in the **AndroidManifest.xml**

Vulnerability	Context	Target	Result	Notes
External Script Accessing localStorage	WebView	localStorage	✔ Successful	Can access sensitive data stored in the **localStorage**
Same-origin Iframe Accessing localStorage	Iframe (file from app origin)	localStorage	✔ Successful	Since the file is loaded from within the app, it can access data stored in **localStorage**
Cross-App File Fetching	WebView	File Created in Another Cordova Application	**X** Blocked	Android sandboxing prevents apps from accessing other apps' files in **/Android/data/...**

5.2.2 Ionic Results

The evaluation of Ionic applications reveals a security posture that differs substantially from that observed in Cordova. Ionic applications benefit from a more modern WebView architecture, improved defaults, and a less permissive bridging model. However, the results also indicate that several vulnerability classes remain exploitable when developers expose Capacitor's runtime objects in the global JavaScript context. Table III summarizes the outcomes of the vulnerability tests performed on representative Ionic applications.

The tests show that, as in Cordova, external script injection is feasible in Ionic applications lacking restrictive Content Security Policies. Once arbitrary JavaScript is executed inside the WebView, it gains full access to Web Storage APIs such as `localStorage` and `sessionStorage`. More importantly, whenever `window.Capacitor` is accessible to web content (which is the

default unless explicitly disabled), the injected script can call sensitive native APIs, including the Capacitor Preferences and Filesystem plugins. This enables an attacker to retrieve data stored by the application or access files written to the device.

In contrast, Ionic Storage (the high-level key-value API provided by the Ionic framework) remains inaccessible to injected scripts unless the developer manually exposes a storage instance to the global JavaScript scope. This indicates that Ionic Storage introduces an additional layer of indirection, which reduces exposure compared to Web Storage or Capacitor plugins, although it is not inherently a security boundary.

Iframe-based tests confirm a stricter adherence to SOP compared to Cordova. Cross-origin iframes are unable to access Web Storage or Capacitor APIs, as expected under the default Chromium security model. However, same-origin iframes such as resources loaded from the application's own `/assets/` directory inherit full access to storage and native plugin interfaces, again provided that `window.Capacitor` is exposed. These results demonstrate that same-origin iframe attacks remain feasible when internal application pages can be manipulated or loaded in unintended contexts.

Further tests show that iframe attempts to load external pages protected by `X-Frame-Options` are correctly blocked, indicating that Ionic's WebView respects modern anti-framing headers. Additionally, privileged operations such as clipboard reading and camera access are reachable through injected scripts, but only if `window.Capacitor` is present and the user grants the required runtime permissions. This confirms that Capacitor's permission model limits abuse to

scenarios in which either permissions have been previously granted or the attacker can induce the user to authorize them.

Overall, the results illustrate that Ionic's security largely depends on the developer's WebView configuration and on whether the Capacitor runtime is exposed globally. While the default platform behavior is more restrictive than Cordova's, insecure CSP settings or the unintentional exposure of `window.Capacitor` can still lead to severe compromise scenarios involving storage access, file retrieval, and invocation of sensitive native APIs.

TABLE III: VULNERABILITIES TESTED IN IONIC.

Vulnerability	**Context**	**Target**	**Result**	**Notes**
External script accessing localStorage	WebView	localStorage, sessionStorage	✔ Successful	Full access
External script accessing Capacitor Preferences	WebView	Capacitor. Plugins. Preferences	✔ Successful	As long as **window.Capacitor** is exposed
External script accessing Ionic Storage	WebView	Ionic Storage	**X** Blocked	Unless the developer exposes **window.storageInstance** manually
External script accessing Capacitor Filesystem	WebView	Capacitor. Plugins. Filesystem	✔ Successful	Can read files created within the app as long as **window.Capacitor** is exposed
Cross-origin iframe accessing localStorage	Iframe (file from remote server)	localStorage, sessionStorage	**X** Blocked	**SOP** blocks access to parent window
Cross-origin iframe accessing Capacitor Preferences	Iframe (file from remote server)	Capacitor. Plugins. Preferences	**X** Blocked	Can't access **window.Capacitor** from different origin
Same-origin iframe accessing localStorage	Iframe (file from /assets/)	localStorage, sessionStorage	✔ Successful	Since the file is loaded from within the app, it can access the **storage**
Same-origin iframe accessing Capacitor Preferences	Iframe (file from /assets/)	Capacitor. Plugins. Preferences	✔ Successful	Since the file is loaded from within the app, it can access the **storage** as long as **window.Capacitor** is exposed
Iframe loading protected page	Iframe (file from remote server)	External URL with X-Frame-Options	**X** Blocked	**X-Frame-Options** header is respected by iframe, so the file is not loaded
Clipboard access	WebView	Capacitor. Plugins. Clipboard	✔ Successful	Works if **window.Capacitor** is exposed and permission is granted
Camera access	WebView	Capacitor. Plugins. Camera	✔ Successful (camera is prompted)	Works if **window.Capacitor** is exposed and permission is granted

5.2.3 React Native Results

React Native applications exhibited the strongest security posture among the three frameworks analyzed. In contrast to Cordova and Ionic, React Native does not rely on a WebView to render its interface, and thus is inherently immune to many WebView-driven attacks unless the developer explicitly embeds a WebView component. The tests summarized in Table IV reflect this architectural advantage: most WebView-based attacks were blocked by default browser security mechanisms or React Native's conservative WebView configuration.

The first set of tests concerned attempts to bypass CORS restrictions. Regardless of whether the request originated from a WebView, a malicious iframe, or a local HTML file, all CORS-protected external APIs correctly rejected unauthorized cross-origin requests. This behavior confirms that React Native's WebView fully adheres to the SOP and does not introduce non-standard extensions or bypasses that could compromise remote endpoints.

Similarly, attempts to load external pages protected with `X-Frame-Options` headers were consistently blocked. The embedded WebView respected the browser's anti-framing directives, preventing malicious iframes from rendering or interacting with protected content. Mixed-content protections also behaved as expected: HTTP resources could not be loaded within an HTTPS WebView unless the developer explicitly weakened the configuration by enabling clear-text traffic at the Android manifest level or setting `mixedContentMode="always"`. This demonstrates that React Native inherits the robust default security properties of modern Chromium-based WebViews.

However, one critical vulnerability class emerged from the exploratory analysis and was confirmed by the structured testing process: the risk associated with forwarding sensitive application state into the WebView environment. In the final test from Table IV, external script injection was successful due to the application's practice of passing an authentication token from `AsyncStorage` to the WebView via `postMessage`. Because the WebView simultaneously permitted arbitrary JavaScript execution and accepted content from unrestricted origins, an attacker-controlled page could read the forwarded token and exfiltrate it to an external server. This unified attack vector discussed in detail in Section 3.2 demonstrates that even though `AsyncStorage` itself is secure at rest, placing its contents inside the WebView context undermines all isolation guarantees.

Importantly, this vulnerability is not inherent to React Native's architecture, but rather rooted in unsafe developer practices. When React Native applications avoid bridging secrets into a WebView and maintain restrictive origin policies, the attack surface remains small and difficult to exploit. But when developers merge native state with permissively configured WebViews, React Native becomes susceptible to the same class of WebView-based attacks affecting Cordova and Ionic.

Overall, the results confirm that React Native provides the safest baseline among the frameworks examined. Most attack attempts, namely, CORS circumvention, anti-framing bypass, mixed-content loading, and iframe escalation were blocked by default. The only successful attack required explicit developer actions that weakened the security model. This highlights the importance of caution when integrating WebViews into React Native applications and re-

inforces the broader conclusion that secure-by-default frameworks can still be compromised through improper design decisions.

TABLE IV: VULNERABILITIES TESTED IN REACT NATIVE.

Vulnerability	**Context**	**Target**	**Result**	**Notes**
Fetch with CORS from WebView	WebView	External API with CORS protection	**X** Blocked	**CORS** is respected
Fetch with CORS from iframe in WebView	Malicious file loaded in iframe	External API with CORS protection	**X** Blocked	**CORS** is respected
Fetch with CORS from local file	Local HTML file	External API with CORS protection	**X** Blocked	**CORS** is respected
Iframe loads protected page	WebView Component	External URL with X-Frame-Options	**X** Blocked	**X-Frame-Options** header is respected by iframe, so the file is not loaded
HTTP loading in WebView	WebView	External website	**X** Blocked	Blocked by default (unless using **Expo React Native** project or **usesCleartextTraffic="true"** in the **AndroidManifest.xml**)
HTTP iframe loading in HTTPS WebView	WebView	External website	**X** Blocked	Blocked by default due to mixed content restrictions, but it can be allowed by using **<WebView mixedContentMode="always" />**
External script	WebView	asyncStorage	✔ Successful	**asyncStorage** token is passed to the WebView via **postMessage**

5.3 Security Analysis of Popular Applications

In addition to the controlled tests performed on mock applications, this thesis evaluated a selection of well-known real-world applications built with each framework. The goal of this analysis was to determine whether the vulnerability patterns identified during exploratory testing and automated scanning were also present in large, widely deployed production apps. For each framework—Cordova, Ionic (Capacitor-based), and React Native—a set of popular applications was identified through official framework showcases, public repositories, developer websites, and third-party listings. These applications were then subjected to the same static and manual testing techniques described in Chapter 3.

The following subsections summarize the results of these evaluations. The tables report whether each application was affected by the vulnerabilities associated with its framework, based on the presence of insecure configurations, exposed plugin interfaces, or unsafe WebView integration patterns.

5.3.1 Cordova Applications

Table V presents the results for the most prominent Cordova applications identified during the dataset collection. The findings reveal a consistent trend: many widely used Cordova apps still expose functionality that can be accessed through injected scripts and same-origin iframes. Well-known applications such as Moodle, Bajaj Finserv, eToro, and Morgan Stanley exhibited complete exposure across all tested dimensions, including access to the File Plugin API, retrieval of local HTML files, and extraction of `localStorage` through both direct script injection and iframe-based attacks.

While some applications showed partial resilience - for example, Western Union and IDBI Bank blocking access to the File Plugin API - most remained vulnerable to at least one of the tested attack paths. These results confirm that insecure configurations in Cordova persist even in high-profile deployments, likely due to legacy design choices and the reliance on permissive defaults.

TABLE V: VULNERABILITY ANALYSIS OF CORDOVA APPLICATIONS.

	File Plugin API		**localStorage**		**HTML file**	
Application	**Script Injection**	**Iframe Injection**	**Script Injection**	**Iframe Injection**	**Script Injection**	**Iframe Injection**
Moodle	✔	✔	✔	✔	✔	✔
Bajaj Finserv	✔	✔	✔	✔	✔	✔
Western Union					✔	✔
Unicredit			✔	✔	✔	✔
IDBI Bank					✔	
Libertex			✔	✔	✔	✔
StormGain			✔	✔	✔	✔
UBA Mobile					✔	✔
FedMobile					✔	
eToro	✔	✔	✔	✔	✔	✔
Banco General			✔	✔		
Morgan Stanley	✔	✔	✔	✔	✔	✔
myCSS			✔	✔	✔	✔
Fanreact	✔				✔	
Ubank Money App			✔	✔	✔	✔
ChefSteps			✔			
CryptoChange			✔	✔		

5.3.2 Ionic Applications

Table VI reports the analysis results for popular Ionic applications built with the Capacitor runtime. Compared to Cordova, Ionic apps displayed fewer critical exposures, although many still exhibited vulnerabilities when plugin interfaces were globally exposed via `window.Capacitor` or when storage mechanisms were left unprotected.

Applications such as Sworkit and Vygo demonstrated broad vulnerabilities across all categories, including access to Capacitor Preferences, localStorage, the Filesystem plugin, and device-level features like the clipboard and camera. Others, such as BHD or Lecturio, restricted some plugin access but remained vulnerable to storage-based attacks. Overall, the results indicate that insecure WebView usage and permissive plugin exposure remain common pitfalls even in modern Ionic/Capacitor applications.

TABLE VI: VULNERABILITY ANALYSIS OF IONIC APPLICATIONS.

	Capacitor Preferences		**localStorage**				
Application	**Script Injection**	**Iframe Injection**	**Script Injection**	**Iframe Injection**	**Capacitor Filesystem**	**Clipboard**	**Camera**
MyBlock			✔	✔	✔	✔	✔
BHD			✔	✔		✔	
Tufts Health Plan			✔	✔	✔		
Sworkit	✔	✔	✔	✔	✔		✔
BCMSM			✔	✔	✔		
AAA Mobile			✔	✔	✔		✔
MyAflac			✔		✔	✔	✔
Walden Univ. Lecturio			✔	✔			
Vygo	✔	✔	✔	✔	✔	✔	✔
JustWatch			✔	✔		✔	

5.3.3 React Native Applications

Finally, Table VII summarizes the results for widely used React Native applications. As expected from the framework's architecture, React Native applications displayed far fewer vulnerabilities than their Cordova or Ionic counterparts. The only vulnerability systematically detected was related to the unsafe practice of forwarding `AsyncStorage` values into a permissively configured WebView. Several major applications, including Discord, Coinbase, Amazon Shopping, and Societe Generale, were found to pass sensitive tokens into WebViews, making them susceptible to the unified AsyncStorage exfiltration attack described earlier.

Other high-profile applications, such as Facebook and Stock Plan, did not exhibit this behavior, highlighting that the vulnerability is tied to developer choices rather than limitations of the framework itself. Overall, the results confirm that React Native provides a more secure baseline, but improper WebView integration can still undermine its security guarantees.

TABLE VII: VULNERABILITY ANALYSIS OF REACT NATIVE APPLICATION.

Application	**asyncStorage**
Untappd	✔
iMobile	✔
Societe General	✔
Stock Plan	
Amazon Shopping	✔
Coinbase	✔
Discord	✔
Facebook	
Mercari	✔
Tesla	✔

5.4 AndroZoo Brute-Force Scan Results

In addition to evaluating known Cordova, Ionic, and React Native applications, a large-scale brute-force scan was conducted on a subset of the AndroZoo repository. The goal of this experiment was to identify hybrid applications "in the wild," without relying on curated lists, public showcases, or framework self-identification. This step provides an unbiased view of the actual distribution of hybrid frameworks within the broader Android ecosystem, serving as external validation of the framework detection module in the static analysis script.

A randomly selected subset of AndroZoo APKs was downloaded and processed through the unified static analysis pipeline. For each APK, the script attempted to determine whether it was built using Cordova, Ionic/Capacitor, React Native, or none of the targeted frameworks. The detection relied exclusively on file-system artifacts and structural markers, as described in Section 3.4.2, allowing the scanner to autonomously classify applications without prior knowledge of their technology stack.

5.4.1 Framework Identification Findings

The brute-force scan revealed that hybrid applications represent only a small fraction of the randomly sampled AndroZoo set. Cordova and React Native were detected more frequently than Ionic/Capacitor, likely reflecting the longer historical prevalence of Cordova and the widespread adoption of React Native among large commercial apps. Ionic/Capacitor apps appeared less frequently, consistent with their more recent emergence and partial overlap with web-first development teams.

A non-negligible proportion of APKs displayed ambiguous or partial indicators—such as residual Cordova assets left after migration to Capacitor, or embedded WebViews used for isolated features without representing a full hybrid architecture. These apps were labeled as non-hybrid by the script, in accordance with the strict classification rules described earlier. Table VIII shows the results of the script execution, while the apps that exhibit at least one vulnerability related to their framework are marked as vulnerable.

TABLE VIII: BRUTE-FORCE SCAN RESULT.

Framework	**Total Apps**	**Vulnerable Apps**
React Native	387	358
Ionic/Capacitor	34	28
Cordova	296	293
Unknown/Other	3664	0
Total	**4381**	**679**

CHAPTER 6

CONCLUSION

This thesis set out to evaluate the security posture of three major cross-platform mobile development frameworks - Cordova, Ionic (Capacitor-based), and React Native - through a combination of exploratory analysis, controlled vulnerability testing, automated static analysis, and targeted dynamic experimentation. The overarching objective was to understand how architectural choices, WebView usage, and bridging mechanisms influence the exposure of hybrid applications to web-origin and native-level attacks.

The exploratory phase and the construction of mock applications established the architectural foundations that shape each framework's security properties. Cordova, the oldest and most legacy-oriented of the three frameworks, exhibited the broadest and most consistent exposure to WebView-driven threats. Weak or missing Content Security Policies, permissive `allow-navigation` and `access-origin` rules, and globally exposed plugin interfaces allowed successful exploitation of multiple vulnerability classes, including script injection, local file disclosure, and persistent storage exfiltration. These weaknesses were also reflected in real-world applications: the large-scale dataset showed that Cordova apps frequently adopt insecure defaults or permissive configurations, which can be detected via static analysis.

Ionic applications, while grounded in a more modern architecture, still retained meaningful attack surfaces. When the global `window.Capacitor` object was exposed, injected JavaScript could access sensitive native APIs such as Preferences, Filesystem, Clipboard, or Camera. De-

spite some improvements—such as strict enforcement of the SOP and the isolation of Ionic Storage—the risk introduced by exposing native capabilities through a WebView context remains substantial. The results indicate that the security of Ionic apps depends heavily on developer-enforced isolation and CSP rigor, rather than on framework-level restrictions alone.

React Native, in contrast, emerged as the most secure framework observed in this study. Unlike Cordova and Ionic, React Native does not rely on WebViews for rendering its interface and exposes native functionality through a strongly typed, structured bridge. As a consequence, most of the vulnerabilities observed in other frameworks simply do not apply. The few risks identified originated not from the framework itself but from application-level decisions—specifically, developers embedding a WebView with permissive settings and forwarding sensitive data (e.g., AsyncStorage values) into it. When used according to its design principles, React Native provides a significantly smaller attack surface.

The brute-force scan of the AndroZoo dataset further supported the external relevance of these findings. Hybrid applications represented only a small subset of the scanned APKs, yet many of those identified exhibited misconfigurations consistent with the vulnerability classes defined in this thesis. This alignment between controlled tests and real-world samples reinforces the claim that hybrid misconfigurations are not merely theoretical concerns, but active and recurring issues in deployed applications.

Dynamic analysis using Frida provided an additional layer of validation. By enabling WebView debugging at runtime, a practical exploitation scenario was demonstrated against a real application (the Moodle Android client). In this case, the combination of a wildcard CSP and

permissive WebView settings enabled JavaScript-based exfiltration of both the `index.html` file and authenticated session cookies to an external server. Although this required a rooted device and access to debugging features that attackers may not always possess, the experiment confirmed the real-world consequences of insufficient WebView hardening and insecure CSP configurations.

Taken together, these findings highlight that hybrid frameworks are powerful and productive development tools, but they also introduce security concerns that require deliberate attention. Most of the vulnerabilities identified are not intrinsic flaws of the frameworks themselves; instead, they stem from configuration choices, assumptions about trust in local WebViews, and insufficient control over injected or dynamically loaded content. Frameworks must therefore offer safer defaults, clearer security guidance, and stronger isolation mechanisms. Likewise, developers must adopt best practices, apply strict CSP rules, and avoid exposing privileged native functionality unnecessarily.

The unified static analysis script developed in this thesis contributes to these goals by providing a reproducible and scalable methodology for detecting common misconfigurations and dangerous framework patterns. Its ability to process large datasets, automatically classify frameworks, and consistently identify security issues can support both research and industry efforts to better understand and mitigate risks in hybrid application development.

6.1 Methodology Limitations

While the methodology adopted in this thesis provides a structured and scalable approach to evaluating hybrid application security, several limitations must be acknowledged to contextualize the results.

Static Analysis Constraints. Static analysis examines applications only in their decompiled, at-rest form. Accordingly, it cannot capture dynamic behaviors that manifest only during execution, such as:

- JavaScript injected at runtime,
- navigation flows triggered by user interaction,
- code paths that depend on remote servers or runtime network conditions,
- dynamically generated, encrypted, or obfuscated JavaScript.

Consequently, static analysis may miss vulnerabilities that activate only under specific runtime circumstances, and it cannot reliably distinguish between reachable and unreachable code paths, which may result in false negatives.

Obfuscation and Minification. Hybrid applications frequently use bundling and optimization tools (Webpack, Rollup, Metro bundler, R8, ProGuard), which can:

- flatten or reorganize asset directories,
- compress JavaScript into monolithic bundles,
- rename symbols and plugin interfaces,

- introduce build-time optimizations that hide code paths.

Although the analysis script is designed to be resilient to common packing strategies, heavy obfuscation may conceal indicators of vulnerable behavior or plugin usage, reducing detection accuracy.

Limitations of the Dynamic Analysis. Dynamic testing with Frida was intentionally narrow in scope. It focused on a single vulnerable real-world application and did not attempt a dataset-wide dynamic assessment. Furthermore:

- enabling WebView debugging generally requires root access,
- Chrome DevTools provides capabilities far beyond those available to real attackers,
- the debugging environment affords visibility and injection opportunities that are unavailable on non-debuggable production devices.

Thus, the dynamic attack confirms that the vulnerability is exploitable under permissive conditions, but does not imply that attackers can necessarily reproduce the same exploit path on arbitrary user devices.

6.2 Future Works

Several directions emerge from this work and could further strengthen the understanding of hybrid application security. A first natural extension would be to scale the static analysis pipeline to a significantly larger portion of the AndroZoo dataset. Although the present study already demonstrates the feasibility of large-scale scanning, analyzing a broader sample would

provide a more complete view of how widespread the identified misconfigurations are in real-world applications.

Another promising line of research involves expanding the study to additional cross-platform frameworks such as Flutter or Kotlin Multiplatform. Because these technologies do not rely on WebView-based architectures, they may expose different security properties and offer a valuable point of comparison with the frameworks examined in this thesis.

Future work may also focus on enhancing the dynamic analysis component. Automating runtime testing—through instrumentation, guided execution, or lightweight fuzzing—could reveal behaviours and vulnerabilities that static analysis alone cannot detect, especially those dependent on user interaction or conditional navigation.

Finally, further refinement of the static analysis techniques could improve detection accuracy in applications that employ heavy obfuscation or bundling. Integrating semantic analysis or pattern-based heuristics may help identify hybrid components even when traditional structural indicators are obscured.

Taken together, these directions highlight the potential for a more comprehensive and in-depth investigation into the security posture of hybrid applications, as well as opportunities to support developers through more robust analysis tools and clearer security guidance.

APPENDICES

Appendix A

FIGURE CREDITS

- Figure 1 - Original image from: https://www.telerik.com/blogs/hybrid-or-native-mobile-app-use-the-right-tool-for-the-job
- Figure 4 - Original image from: `https://digimonksolutions.com/mobile-app-architecture/`
- Figure 5 - Original image from: https://bosctechlabs.com/react-native-new-architecture-in-2023/
- Figure 6 - Original image from: https://medium.com/riseconsulting/microsoftun-mobil-uygulama-geli
- Figure 7 - Original image from: https://securityboat.net/deep-dive-into-android-security/

CITED LITERATURE

1. DevonBlog: Difference between CORS and CSP security headers. DevonBlog, n.d. [Online, accessed 02/11/2025].

2. Apache Cordova Documentation: https://cordova.apache.org/docs/en/latest/index.html, n.d. [Online, accessed 02/25/2025].

3. Intel Forum: https://community.intel.com/t5/software-archive/cordova-webview-vs-inappbrowser/m-p/1076167, 2017. [Online, accessed 03/06/2025].

4. Beer, P., Squarcina, M., Veronese, L., and Lindorfer, M.: Tabbed out: Subverting the Android custom tab security model. In 2024 IEEE Symposium on Security and Privacy (SP), pages 4591–4609. IEEE, May 2024.

5. Yang, G., Huang, J., and Gu, G.: Iframes/Popups are dangerous in mobile WebView: Studying and mitigating differential context vulnerabilities. In 28th USENIX Security Symposium (USENIX Security 19), pages 977–994, Santa Clara, CA, August 2019. USENIX Association.

6. Ionic: Ionic documentation. `https://ionicframework.com/docs`. Ionic Website, n.d. [Online, accessed 03/18/2025].

7. Lynch, M.: Capacitor vs. Cordova: Modern hybrid app development. Ionic Resources, 2022. [Online, accessed 03/20/2025].

8. Goyal, R.: Basic security for Ionic & Cordova mobile applications. `https://medium.com/@rohit157.rg/basic-security-for-ionic-cordova-mobile-applications-d7a9f06c767b`, August 2020. [Online, accessed 27/11/2025].

9. LinkedIn Community: https://www.linkedin.com/advice/1/what-pros-cons-capacitor-vs-cordova-native-features, n.d. [Online, accessed 03/23/2025].

10. Meta Platforms, Inc.: React native documentation. `https://reactnative.dev/docs/getting-started`, n.d. [Online, accessed 05/17/2025].

CITED LITERATURE (continued)

11. Sattlegger, P. F.: Security analysis of WebViews in cross-plattform mobile frameworks. Master's thesis, Technische Universität Wien, 2023. [Online] Available: https://repositum.tuwien.at/handle/20.500.12708/188813.

12. Illogical Robot LLC: Apkmirror. https://www.apkmirror.com/, n.d. [Online, accessed 08/28/2025].

13. Capgo: Top Cordova apps. https://capgo.app/top_cordova_app/, n.d. [Online, accessed 08/28/2025].

14. APKPure: Apkpure. https://apkpure.com/, n.d. [Online, accessed 08/28/2025].

15. Androzoo: Androzoo documentation. https://androzoo.uni.lu/api_doc, 2016. [Online, accessed 09/03/2025].

16. Ionic: Ionic showcase. https://ionic.io/showcase, n.d. [Online, accessed 09/24/2025].

17. Google Developers: Remote debugging WebViews https://developer.chrome.com/docs/devtools/remote-debugging/webviews, n.d. [Online, accessed 10/04/2025].

18. Kumar, T.: Android webview hacking: Enable webview debugging. https://infosecwriteups.com/android-webview-hacking-enable-webview-debugging-d292b53f7a63. InfoSec Write-ups, September 2020. [Online, accessed 10/06/2025].

19. n1sh1th: Frida script to enable WebView debugging. https://gist.github.com/n1sh1th/d3ccd68ee17eb9b94b28ecfd6514854b, 2021. [Online, accessed 10/06/2025].

20. Ostorlab Team: Testing Cordova applications. https://blog.ostorlab.co/testing-cordova-applications.html, November 2016. [Online, accessed 10/19/2025].

21. Wayal, V.: Effortless pentesting of apache Cordova applications. https://payatu.com/blog/effortless-pentesting-of-apache-cordova-applications/, July 2023. [Online, accessed 10/10/2025].

VITA

NAME	Federico Civitareale
EDUCATION	
	Master of Science in “Computer Science”, University of Illinois at Chicago, Dec 2025, USA
	Specialization Degree in “Cybersecurity”, Dec 2025, Polytechnic of Turin, Italy
	Bachelor’s Degree in ”Computer Engineering”, Mar 2023, Università degli Studi dell’Aquila, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	A.Y. 2024/25 - One Year of study abroad in Chicago, Illinois
	A.Y. 2023/24 - Lessons and exams attended exclusively in English
	2023 - TOEFL Examination (101/120)
	2018 - Cambridge C1 Advanced Certification (180/210)
Spanish	Basic understanding
SCHOLARSHIPS	
Fall 2024	Italian scholarship for TOP-UIC students
TECHNICAL SKILLS	
Programming Languages	Python, JavaScript, Bash, Java, PHP
Frameworks	Git, Docker, Kubernetes, Node.js, React.js, SQL
Cybersecurity Tools	Burp Suite, Nmap, TCPdump, Pwntools, OpenSSL, Scapy, Wireshark

VITA (continued)

Skills	Software Engineering, Artificial Intelligence, Neural Networks, Databases, Data Management, Programming & Software Development, Academic Research, Web Application Development, System Design, Object-Oriented Programming

ACADEMIC PROJECTS

2024- 2024	Web Application Project
	Developed a web-based ticketing platform with real-time updates and ticket categorization. Utilized JavaScript, Node.js, and React for the interface, and Passport.js for secure session management.
2023 - 2023	Patterns in Encrypted Web Traffic
	Implemented machine learning models to classify TCP flows and identify server origins. Applied clustering and regression techniques to analyze and understand server behavior within encrypted web traffic.
2022 - 2023	Iterative Algorithms on Apache Spark
	Implemented iterative algorithms using PySpark and NumPy to solve large systems of linear equations. Optimized scalability on a 20-node cluster, significantly reducing execution time through Spark's parallel processing.