Umeå University

Master Thesis

From Ignition Interlock Servicing to SAST: Effective Taint Analysis for .NET MAUI Blazor Hybrid Applications

Author:
Miles Lundqvist

Supervisor:
Monowar Bhuyan

Examinator:
Ola Ringdahl

External supervisor:
Wilhelm Ackermann

Submitted for the Degree of
Master of Science in Interaction Technology and Design, 300ECTS
Department of Computer Science

June 9, 2025

Abstract

Hybrid mobile application frameworks, such as .NET MAUI Blazor, make it more efficient to develop cross-platform applications. Still, they introduce some unique challenges, such as accurately tracing data flows across language boundaries and modeling framework-specific component lifecycles, that existing security analysis tools do not sufficiently address. This thesis explores the efficacy of current open-source security testing tools in identifying vulnerabilities within .NET MAUI Blazor applications on Android, aligned with the OWASP Mobile Application Security Verification Standard (MASVS).

A vulnerable .NET MAUI Blazor Hybrid application named *IVMAUIB* is developed as a testbed for evaluating the effectiveness of existing security tools. This benchmark application incorporates various weaknesses that align with the MASVS. Five open-source static and dynamic security analysis tools—MobSF, OWASP ZAP, SonarQube Community, Xamarin Security Scanner, and Security Code Scan—are assessed using IVMAUIB. Additionally, a prototype static taint analysis tool called *FlowMauiBlazor* is developed utilizing the Roslyn compiler platform and implemented in C#. FlowMauiBlazor employs the Interprocedural Finite Distributive Subset (IFDS) framework specifically targeting .NET MAUI Blazor applications. It focuses on tracking taint propagation, particularly identifying sources and sinks related to C# and JavaScript interoperability.

Results indicate existing tools effectively detect generic vulnerabilities but often miss Blazor-specific issues. FlowMauiBlazor demonstrates high precision in detecting cross-boundary taint flows and insecure local storage vulnerabilities that generic tools overlook. FlowMauiBlazor explicitly models the .NET MAUI Blazor framework, including its component lifecycles and JavaScript interoperability mechanisms. This highlights the need for specialized analysis tools tailored to .NET MAUI Blazor frameworks.

Future work includes expanding prototype capabilities with broader MASVS coverage, improving framework modeling, and integrating JavaScript analysis.

Keywords: .NET MAUI Blazor, Hybrid Mobile Security, Static Analysis, IFDS, OWASP MASVS, Roslyn

Acknowledgements

I want to express my sincere gratitude to those who supported me throughout the completion of this thesis.

Firstly, I thank Advania for having me and my external supervisor, Wilhelm Ackermann, for the support and guidance throughout the project. I also want to thank the entire team at the office for the much-needed breaks that keep me energized and motivated.

I am equally thankful to Monowar Bhuyan, my internal supervisor at Umeå University, whose expertise and constructive feedback have contributed to this research.

Additionally, I wish to thank my peer review group for their helpful discussions, encouragement, and insightful comments that helped refine my work.

Finally, I am grateful to my family and friends for their constant encouragement throughout this journey. I am especially thankful to my girlfriend for her patience and support during both this research process and our years of study together.

Contents

List of Figures

List of Tables

List of Abbreviations

IID	**I**gnition **I**nterlock **D**evice
BAC	**B**lood **A**lcohol **C**oncentration
CA	**C**ertificate **A**uthority
OWASP	**O**pen **W**eb **A**pplication **S**ecurity **P**roject
MASVS	**M**obile **A**pplication **S**ecurity **V**erification **S**tandard
MASTG	**M**obile **A**pplication **S**ecurity **T**esting **G**uide
SDK	**S**oftware **D**evelopment **K**it
SAST	**S**tatic **A**pplication **S**ecurity **T**esting
DAST	**D**ynamic **A**pplication **S**ecurity **T**esting
IFDS	**I**nterprocedural **F**inite **D**istributive **S**ubset
CVE	**C**ommon **V**ulnerabilties and **E**xposures
CWE	**C**ommon **W**eakness **E**numeration
CFG	**C**ontrol **F**low **G**raph
ICFG	**I**nterprocedural **C**ontrol **F**low **G**raph

Chapter 1

Introduction

Driving under the influence (DUI) remains a critical public safety issue worldwide, leading to substantial fatalities, injuries, and property damage each year. In the United States alone, impaired driving contributed to 13,424 deaths in 2022 [1, 2], with approximately 1 million DUI-related arrests made annually [1, 3]. Similarly, in Sweden, 229 traffic-related fatalities were reported in 2023, of which 52 were attributed to alcohol- or drug-related incidents [4]. These figures underscore the need for effective prevention and intervention strategies to reduce DUI recidivism and enhance road safety. One technological intervention that has proven effective is the ignition interlock device (IID), a breathalyzer-like device installed in vehicles to prevent engine start if the driver's blood alcohol concentration (BAC) exceeds a set limit [5–7]. IIDs are widely used in both individual and commercial contexts [8], and evidence suggests that IID systems can be highly effective in reducing incidents of impaired driving [6]. However, maintaining these devices requires regular servicing, including data log extraction, firmware updates, and configuration changes, typically performed at authorized service centers [6]. This traditional servicing model introduces logistical challenges, especially for commercial fleets and users in remote areas.

To address these challenges, a proof-of-concept mobile application for remote IID servicing was developed as the initial part of this thesis project. This Android-based hybrid application, built using .NET MAUI Blazor, aimed to facilitate remote service of IIDs. Although promising in terms of usability and operational flexibility, this development highlighted potential new cybersecurity risks: a compromised smartphone could expose vast amounts of personal data [9] or allow malicious actors to bypass or tamper with critical IID functions, undermining the device's purpose. While the IID servicing prototype was not subjected to the same formal evaluation as the benchmark application developed later in this research, the process of its initial construction revealed a broader issue: existing security tools struggle to analyze complex hybrid frameworks like .NET MAUI Blazor effectively. Prior research has shown that hybrid apps, which combines web and native code, lead to intricate interactions that many analysis tools fail to handle comprehensively [10–12].

This observation led to a shift in focus; rather than evaluating the security of a specific application, this thesis investigates the effectiveness of existing security analysis tools when applied to MAUI Blazor hybrid apps. A separate, vulnerable benchmark application, *IVMAUIB*, was created to facilitate reproducibility and empirical testing. IVMAUIB was designed to include representative weaknesses across the categories defined by the OWASP Mobile Application Security Verification Standard (MASVS) [9]. It serves as a controlled testbed for assessing existing open-source tools. Furthermore, a custom static analysis prototype, hereafter referred to as *FlowMauiBlazor*, was

built based on the Interprocedural Finite Distributive Subset (IFDS) framework [13]. FlowMauiBlazor, implemented with C# using Roslyn [14], aims to improve taint flow detection precision and coverage for .NET MAUI Blazor applications and was evaluated against IVMAUIB.

1.1 Objectives and Research Questions

This thesis investigates the current state of security analysis for .NET MAUI Blazor Hybrid applications on Android. It aims to assess how well current open-source security tools align with the OWASP MASVS and to evaluate the potential of FlowMauiBlazor, a prototype Roslyn-based IFDS taint analyzer, in addressing identified gaps in vulnerability detection.

To achieve these aims, the study is guided by the following specific objectives and corresponding research questions:

Objectives

- **O1** – *Benchmark creation:* Build and publish a vulnerable MAUI Blazor Hybrid application that implements representative weaknesses across the MASVS categories (Storage, Crypto, Auth, Network, Platform, Code, Resilience, Privacy).
- **O2** – *Baseline assessment:* Run five open-source static and dynamic analysis tools (MobSF, ZAP, SonarQube, Xamarin Security Scanner, Security Code Scan) on the benchmark and quantify their true-positive, false-positive, and false-negative rates per MASVS category.
- **O3** – *Prototype development:* Design and implement FlowMauiBlazor, with domain-specific source/sink and entry point modelling for MAUI Blazor.
- **O4** – *Comparative evaluation:* Measure FlowMauiBlazor's vulnerability-detection precision, recall, and F1-score on the same benchmark, and compare the results with the static baseline tools.
- **O5** – *Gap analysis and recommendations:* Identify MASVS areas that remain weakly covered, discuss the impact of FlowMauiBlazor's current limitations, and propose future research and tooling directions.

Research Questions

RQ1 How effective are existing open-source static and dynamic security-testing tools at detecting MASVS-aligned vulnerabilities in a .NET MAUI Blazor Hybrid Android application?

RQ2 To what extent does *FlowMauiBlazor*, a prototype IFDS taint analyzer, increase MASVS vulnerability-detection coverage compared with existing static analyzers for .NET MAUI Blazor Hybrid applications?

1.2 External Partner

The research will be conducted in cooperation with the company Advania, a tech company operating in Sweden, Norway, the UK, Iceland, Finland, and Denmark[1].

[1] https://www.advania.com/about

They operate in sectors such as Managed Services, Hardware, and Software. This study was conducted on-site at their Umeå office.

1.3 Thesis Organization

The remainder of this thesis is structured as follows:

- **Chapter 2 – Background:** Introduces the thesis's essential technical and conceptual foundations, including ignition interlock servicing, mobile application security standards such as OWASP MASVS, and the .NET MAUI Blazor hybrid framework. It also presents an overview of security testing tools and the IFDS framework for static analysis.
- **Chapter 3 – Previous Works:** Reviews prior research on hybrid mobile application security, highlighting known challenges in static analysis and tool limitations. This chapter positions the current work within the broader research landscape.
- **Chapter 4 – Methodology:** Describes the methodology used in the study, including designing and implementing IVMAUIB, using open-source security testing tools, and developing FlowMauiBlazor, the IFDS-based static analysis prototype. Technical details regarding IFDS usage and Roslyn integration are also covered.
- **Chapter 5 – Result:** This chapter presents the findings from the empirical evaluations. It includes information on the vulnerabilities implemented in the benchmark application, the detection results from the assessed open-source security tools, and the performance of FlowMauiBlazor on both the benchmark application and the IID service application prototype.
- **Chapter 6 – Discussion:** Presents and discusses the results of running the existing tools and FlowMauiBlazor on the benchmark application. This includes comparing detection performance across MASVS categories and discussing FlowMauiBlazor's implementation, design trade-offs, and limitations.
- **Chapter 7 – Conclusion:** Summarizes the study's key findings, answers the research questions and presents some limitations of FlowMauiBlazor.
- **Chapter 8 - Future Work:** Discusses opportunities for extending this research, suggesting specific avenues to create a more comprehensive and robust static analysis solution for hybrid mobile applications.

Chapter 2

Background

This chapter lays the foundation for understanding the research presented in this thesis. The chapter serves as an introduction to the research presented in this thesis. It starts by explaining the servicing of Ignition Interlock Devices (IIDs) to provide context and demonstrate the necessity for enhanced security analysis in modern hybrid mobile applications.

Next, it outlines essential standards for assessing mobile security, specifically the OWASP Mobile Application Security Verification Standard (MASVS) and the Mobile Application Security Testing Guide (MASTG). The chapter then focuses on key technologies related to this study, which include the Android operating system, the .NET MAUI cross-platform framework, and the Blazor Hybrid model.

Lastly, the chapter offers an overview of tools used for mobile security testing. It introduces basic concepts related to static analysis, such as taint analysis and the Interprocedural Finite Distributive Subset (IFDS) framework.

2.1 Ignition Interlock Servicing

This section defines the meaning of servicing an IID and identifies the software-servicing tasks that are relevant for remote service.

Under the current servicing model, IID users must visit an authorized service center, where technicians establish a physical connection between the device and a computer using a dongle and cable, as illustrated in Figure 2.1. Once connected, the service software communicates with the IID and executes a series of commands. One key task is *calibration checks*, a routine procedure that verifies the accuracy of alcohol-level measurements.[1] Calibration requires specialized equipment and therefore falls outside the scope of this thesis.

The focus of the proof-of-concept developed for this study is on the *software-servicing* aspects of IID maintenance—namely, *data log extraction*, *firmware updates*, and *configuration changes*. These tasks can be performed without requiring users to travel to a service center, making them candidates for remote service solutions. The following subsections describe each task and the challenges of executing them remotely.

[1] Intoxalock FAQ, SmartStart FAQ

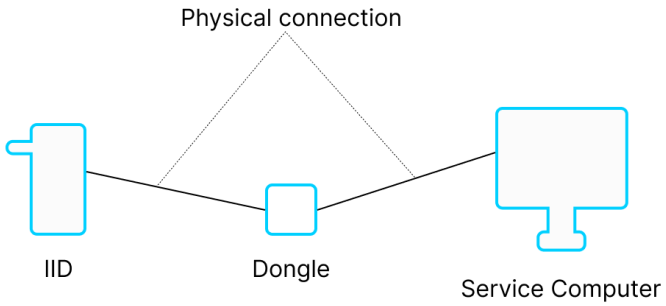

FIGURE 2.1: Current IID Servicing Setup

2.1.1 Data Log Extraction

IIDs maintain internal memory that records event data such as engine start attempts, breath test results, and tampering incidents. These logs are relatively small and require minimal storage. When a tampering attempt, or any bypass effort is detected, the incident is logged. Accumulated violations can lead to additional service requirements before the user is allowed to continue operating the IID.

Data logs are only retrieved during in-person servicing: the service computer connects to the IID, extracts the logs, and uploads them to a central database for further analysis and display. In a remote service approach, the challenge lies in securely storing and transmitting these logs while ensuring data confidentiality and integrity.

2.1.2 Configuration Changes

Adjusting device parameters is another aspect of IID servicing. For example, legal BAC thresholds vary by region; in Sweden, a BAC level below 0.2 g/L is considered sober[2]. Other configurable options include date and time settings, device language, or the internal timer controlling how long the IID can go without being serviced. Under conventional servicing, authorized technicians can only modify these parameters with direct access to the IID.

A remote-service approach must ensure that only legitimate parties can alter settings. Otherwise, a malicious user might increase allowable BAC thresholds or disable lockout features.

2.1.3 Firmware Updates

Firmware is a specialized software program embedded within hardware that dictates device functionality [15]. Like all software, firmware can contain bugs or require enhancements over time to address security vulnerabilities, improve performance, or add features [16]. Updating this software is important for maintaining safe and reliable IID operation.

In the current workflow, firmware updates typically involve taking the IID to a service center, connecting it to a computer, and manually loading the new firmware. Because the IIDs in this study lack native Internet connectivity, they cannot receive updates over the air like typical IoT devices [16, 17].

[2] https://etsc.eu/issues/drink-driving/sweden/

2.2 Open Web Application Security Project

The Open Worldwide Application Security Project (OWASP) is a nonprofit foundation that works to improve the security of software community-led open-source projects. The project foundation aims to help organizations develop, operate, and maintain trustworthy applications [18].

OWASP has many different projects that have been built by the community. One of which is their flagship project called OWASP Mobile Application Security, which consists of a security standard for mobile applications called OWASP MASVS and a thorough testing guide called OWASP MASTG, which can be used to cover the processes, techniques, tools, and test cases that enable testers to deliver consistent and complete results [19].

2.2.1 OWASP MASVS

The OWASP Mobile Application Security Verification Standard (MASVS) serves as a security benchmark, aiding developers in creating, testing, and maintaining secure mobile applications [9]. When this study was conducted, the standard was at version 2.1.0. MASVS is designed to address security challenges in mobile platforms and provides a structured guide for organizations aiming to implement robust security practices.

The standard categorizes security controls into domains, each targeting areas of the mobile application attack surface. A comprehensive list of these control points and their descriptions is provided in Table 2.1.

The MASVS is designed to be platform-agnostic, meaning its guidelines and best practices apply to all types of mobile applications, including native, cross-platform, and hybrid apps [9]. This makes its principles relevant to developing applications with .NET MAUI Blazor Hybrid. This study uses the MASVS categories and control points as the primary criteria for defining, implementing, and evaluating security vulnerabilities [9].

2.2.2 OWASP MASTG

The OWASP Mobile Application Security Testing Guide (MASTG) is a manual for testing the security of mobile applications [20]. It maps to the same set of security requirements defined in the MASVS. The MASTG provides the technical processes and checklists for systematically evaluating those requirements.

Given the number of mobile app frameworks available, it would be impossible to cover them exhaustively. Therefore, the focus of MASTG is on native applications. However, the same techniques are also helpful when dealing with web or hybrid apps. Ultimately, no matter the framework, every app is based on native components [20].

2.3 Development Platform and Technologies

This section presents the development platform and technologies chosen for and relevant to this project, focusing on those that facilitate the creation of hybrid mobile applications like IVMAUIB and the IID servicing prototype.

2.3.1 Android Operation System

The Android operating system is the target platform for the mobile applications investigated and analyzed in this thesis, including the initial IID servicing prototype and IVMAUIB built with .NET MAUI Blazor. Understanding its architecture and security model is therefore essential. Android stands out among the available smartphone platforms due to its significant global market share of around 72% [21, 22], making it a practical choice to ensure widespread compatibility and adoption. Android is also an open-source operating system built on a modified Linux kernel [23], which provides flexibility and extensive community support. The Android OS utilizes Linux user separation to sandbox apps, which means that each app is assigned a unique user ID and runs as a separate process. As a result, by default, apps cannot access each other's data [24]. This implements the principle of least privilege, which means that an app can only access system features or data for which it has explicit permission. Android's way of handling permissions requires apps to declare in their manifest what sensitive resources (e.g., camera, GPS, contacts) they need, and the user must grant these permissions [24].

Android apps have four main component types: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers* [24]. These components are entry points through which the system or other apps interact with an app. Developers specify in the app's manifest file whether a component should be exposed and what protection it has. Malicious apps could invoke or query their data if components are exported without proper access controls. For example, a malicious application might deceive a user into interacting with it while believing they are using the victim application, as outlined in CWE-926 [25].

Above the Linux kernel, Android includes a hardware abstraction layer (HAL) that enables developers to interact with hardware-specific functionalities through standardized interfaces [23]. This abstraction is essential for accessing device components, including USB ports, without directly managing the complexities of hardware drivers. Android's support for USB host mode is particularly relevant to this study, as it allows an Android device to act as a host [26]. This allows connection with external devices, such as potentially IIDs. The Android device can recognize the IID and facilitate two-way data communication over a serial connection through this capability. This built-in functionality simplifies the process of enabling the smartphone to perform remote servicing of the IID.

2.3.2 .NET MAUI

.NET Multi-platform App UI (MAUI) is Microsoft's cross-platform framework for building native mobile and desktop applications with a single codebase using C#. MAUI extends the capabilities of *Xamarin.Forms*, which was originally designed for mobile platforms, by adding support for desktop environments such as Windows and macOS [27]. Hybrid applications combine native and web-based development methods, allowing developers to reuse existing knowledge. This streamlines the development process and supports a unified software architecture [28]. This benefit is especially evident in frameworks such as .NET MAUI, which utilize these principles to simplify cross-platform development.

MAUI applications run within a native container on each platform, typically using a WebView control to render web content when needed [29]. On Android, iOS, and macOS, MAUI utilizes the Mono runtime. A high-level overview of .NET MAUI's

architecture is shown in Figure 2.2. Additionally, MAUI provides a rich collection of controls and native APIs to access device features such as GPS, network state, and sensors [27].

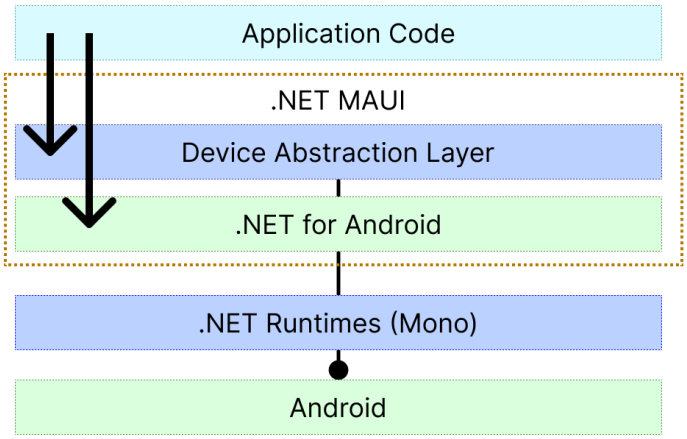

FIGURE 2.2: .NET MAUI Architecture, inspired by Microsoft[27].

2.3.3 Blazor and Blazor Hybrid in .NET MAUI

Blazor is a web UI framework in the .NET ecosystem that allows developers to build interactive web UIs using C#, Razor syntax, and HTML instead of JavaScript [30]. In Blazor, the user interface is built from reusable components, representing entire pages or different controllers [30]. Initially, Blazor offered two hosting models: Blazor Server, where the UI updates are processed on the server over a SignalR connection, and Blazor WebAssembly, where the entire application runs in the browser as WebAssembly [30]. Blazor enables developers with .NET expertise to create single-page applications without switching to JavaScript-based frameworks such as React or Angular [31].

More recently, Blazor has been extended to build native client apps through a hybrid approach. In Blazor Hybrid applications, Razor components run within the native app alongside other .NET code, rendering the user interface as HTML and CSS inside an embedded WebView control. Communication between the web UI and the native code occurs through a local interop channel, enabling seamless integration of web technologies within a native environment [32]. This hybrid model allows developers to leverage their web development skills while accessing native device features through MAUI.

A key aspect of Blazor is the interoperability between the C# .NET code and the JavaScript environment of the WebView. Communication from .NET to JavaScript is typically achieved using the `IJSRuntime` abstraction. JavaScript running in the WebView can, in turn, also invoke .NET methods exposed from C# by decorating them with the `[JSInvokeable]` attribute [33].

2.3.3.1 Component Lifecycle

The Razor component lifecycle is a fundamental concept in Blazor applications, governing how components are created, rendered, and destroyed. Blazor components progress through a series of synchronous and asynchronous lifecycle methods, such as `OnInitializedAsync`, `OnParametersSetAsync`, and `Dispose` [34].

These methods provide hooks for developers to execute application code at specific points in a component's existence. It is possible to use these to include initializing component state, for example, fetching data, responding to changes in component parameters, manually triggering renders, and cleaning up resources when a component is removed from the UI [34].

2.3.3.2 Event Handling and The `@bind` Directive

Blazor components handle user interactions through event handling. DOM events, such as button clicks (`@onclick`) or input changes (`@oninput`), can be bound to C# methods, known as event handlers. When an event occurs, the corresponding C# method is executed, allowing the application to respond to user actions [35].

The `@bind` directive in Blazor is a specific form of event handling that provides a way to achieve two-way data binding between UI elements such as input fields and component properties or fields [36]. This means that user changes in a UI element automatically update the corresponding C# variable. Conversely, programmatic changes to the C# variable are reflected in the UI element.

The basic syntax for the `@bind` directive is:

```
<input @bind="CurrentValue" />
```

By default, `@bind` updates the C# variable when the UI element loses focus (typically using the HTML element's `onchange` event). The Razor compiler transforms these directives and event bindings into C# code, including event handler delegates and logic to update the bound variables.

2.4 Security Testing Tools

Various open-source mobile security testing tools exist to help researchers and analysts identify vulnerabilities and conduct penetration tests on mobile applications. These tools generally fall into two main categories: *static analysis* and *dynamic analysis*, each providing distinct insights into an app's security posture [37].

Static analysis involves scanning an application's source code or compiled binary without executing it [37]. This method efficiently analyzes large amounts of code and can highlight potential security flaws [38]. However, static analysis alone is often insufficient, as it may generate false positives and false negatives [37], requiring manual review by an analyst. Rather than fully automating vulnerability detection, static analysis tools typically assist security professionals by narrowing down areas of interest within the codebase.

On the other hand, dynamic analysis requires running the application and observing its behavior at runtime [37]. This approach efficiently identifies issues related to authentication, authorization, server configuration, and data flow [11]. Dynamic analysis tests real-world behavior by interacting with backend systems and monitoring their responses, unlike static analysis, which only identifies potential flaws by analyzing the code [39].

This section introduces the open-source tools selected for this study, which represent both static and dynamic analysis techniques.

2.4.1 Mobile Security Framework

Mobile Security Framework (MobSF) is an all-in-one, open-source penetration testing platform supporting Android, iOS, and Windows applications [40]. It provides capabilities for both static analysis (decompiling binaries and inspecting code) and dynamic analysis (instrumenting an emulator to observe runtime behavior). Previous research has shown that MobSF effectively detects vulnerabilities in mobile apps [11, 41, 42]. By consolidating multiple analysis techniques into a single tool, MobSF simplifies the process of uncovering insecure configurations and coding practices. Figure 2.3 illustrates the use of MobSF to scan the intentionally vulnerable application *allsafe.apk* [43].

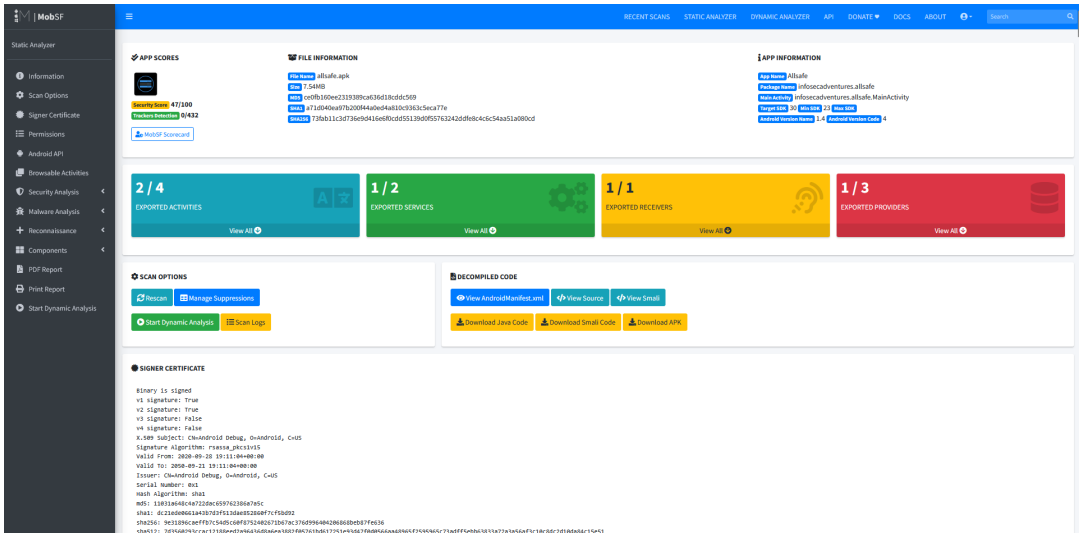

FIGURE 2.3: Result of MobSF statical scan using the intentionally vulnerable allsafe.apk [43]

2.4.2 ZAP

OWASP ZAP (Zed Attack Proxy) is an open-sourced penetration testing tool. It functions as a "man-in-the-middle" proxy, sitting between the analyst's browser and the web application. This allows it to intercept and inspect the messages exchanged between the browser and the web application, modify the contents, and also forward packets either to the original destination or to a new one [44].

2.4.3 Xamarin Security Scanner

Xamarin-Scanner is a static analysis tool designed to detect security vulnerabilities in Xamarin applications. As mentioned in 2.3.2, Xamarin is a cross-platform mobile development framework that is the predecessor to .NET MAUI, including MAUI Blazor, which means Xamarin-based tools remain useful for analyzing projects with shared architectural roots [45].

2.4.4 Security Code Scanner

Security Code Scanner is another static application security testing (SAST) tool created to detect security vulnerabilities in .NET and C# applications. Integrating into

the development workflow enables developers to identify and remediate potential security issues early in the software development lifecycle. This tool can be installed as a Visual Studio extension, a NuGet package, or a stand-alone runner [46].

2.4.5 SonarQube Community

SonarQube Community Edition is an open-source platform developed by SonarSource. Its primary use is to continuously inspect code quality. It supports static code analysis for multiple languages, including C#. It can help developers find bugs, code smells, and security "hotspots" during development [47]. Security hotspots refer to security-sensitive areas of code that require further examination by the user [48].

2.5 Categorization of Vulnerabilities

This study uses MASVS control points to group vulnerabilities. However, to complement these general control points, vulnerabilities can be categorized using publicly known vulnerability databases. Categorizing vulnerabilities helps security researchers and analysts to highlight potential security risks, assess the impact of vulnerabilities, and provides a standardized vocabulary [49]. It also allows entities to evaluate the risk associated with a particular vulnerability, which can guide the priority of creating patches [50].

2.5.1 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a database for publicly disclosed vulnerability identifications [51]. This database is maintained by the MITRE Corporation[3], a non-profit that serves to advance security.

CVE records are standardized identifiers for publicly known cybersecurity vulnerabilities. Each record includes a unique ID, such as *CVE-2014-0160* - which typically contains the year of disclosure, a sequential number, and a brief description. To provide more detailed explanations, fix information, and CVSS scores, NIST maintains the National Vulnerability Database (NVD) [51]. Access to updated information about threats helps organizations keep their systems secure and up-to-date.

Figure 2.4 shows an example of a CVE entry in the NVD, illustrating the structure and available metadata for a known vulnerability[4].

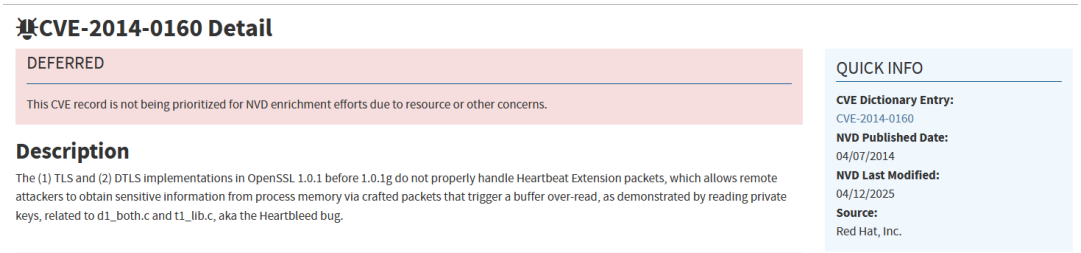

FIGURE 2.4: Screenshot from the NVD showing details for CVE-2014-0160

[3] https://www.mitre.org/who-we-are
[4] https://nvd.nist.gov/vuln/detail/cve-2014-0160

2.5.2 Common Weakness Enumeration

The Common Weakness Enumeration (CWE) complements CVE by focusing on the types of weaknesses or vulnerabilities present in software. While CVE identifies specific vulnerabilities, CWE categorizes the common flaws that can lead to vulnerabilities [52]. This list of software and hardware weaknesses is also community-developed and maintained by MITRE. Developers must understand the weaknesses that lead to vulnerabilities, as it is easier to address these issues before a product is finalized and deployed. The list of CWEs is more stable than the CVE database and receives updates with new information three to four times a year [53].

2.6 Foundational Concepts for Security Analysis

This section presents the foundation for the security analysis conducted in this thesis by introducing two key concepts. First, *taint analysis*, a dynamic or static technique used to identify security vulnerabilities by tracking untrusted data flow through a program. Next, the *Interprocedural, Finite, Distributive, Subset (IFDS) framework* is presented, which provides a formal methodology for performing precise and efficient dataflow analyses, such as taint analysis, across procedural boundaries.

2.6.1 Taint Analysis

Taint Analysis is a security analysis technique used to identify potential vulnerabilities within software by tracking the flow of external data. This analysis begins by identifying *taint sources*, typically methods that receive data from external sources, such as user input fields or external APIs [54]. When data is retrieved from these external sources, it is initially labeled as *tainted* [54], for example, *tainted(x)*.

Once the data has been marked, the analysis tracks the propagation of this tainted data through the program. This involves following the data as it moves through different flows, such as assignments, computations, and method calls, monitoring how tainted data spreads to previously untainted portions of the application [54].

Another aspect of Taint Analysis is the identification of *sinks*. Sinks are locations within the program where tainted data could be misused or lead to vulnerabilities if not handled correctly [54]. Examples of sinks include operations that store sensitive information insecurely, such as saving a password in plaintext on the device.

2.6.2 The IFDS Framework for Interprocedural Dataflow Analysis

The interprocedural finite distributive subset (IFDS) framework provides an efficient, precise, context-sensitive, and flow-sensitive algorithm for solving a specific class of dataflow analysis problems. It applies to many interprocedural problems where the dataflow information can be represented as subsets of a finite domain and the analysis function is distributive [55].

Developed initially by Reps, Horwitz, and Sagiv [13], the IFDS framework models the program using a structure known as a *supergraph*, denoted $G^* = (N^*, E^*)$. This supergraph conceptually combines the control-flow graphs (CFGs) of all procedures (e.g, methods) within the program. Each procedure p has a corresponding flow graph G_p with a unique start node s_p and a unique exit node e_p. The nodes within each G_p represent the individual states and predicates of that procedure. A procedure call

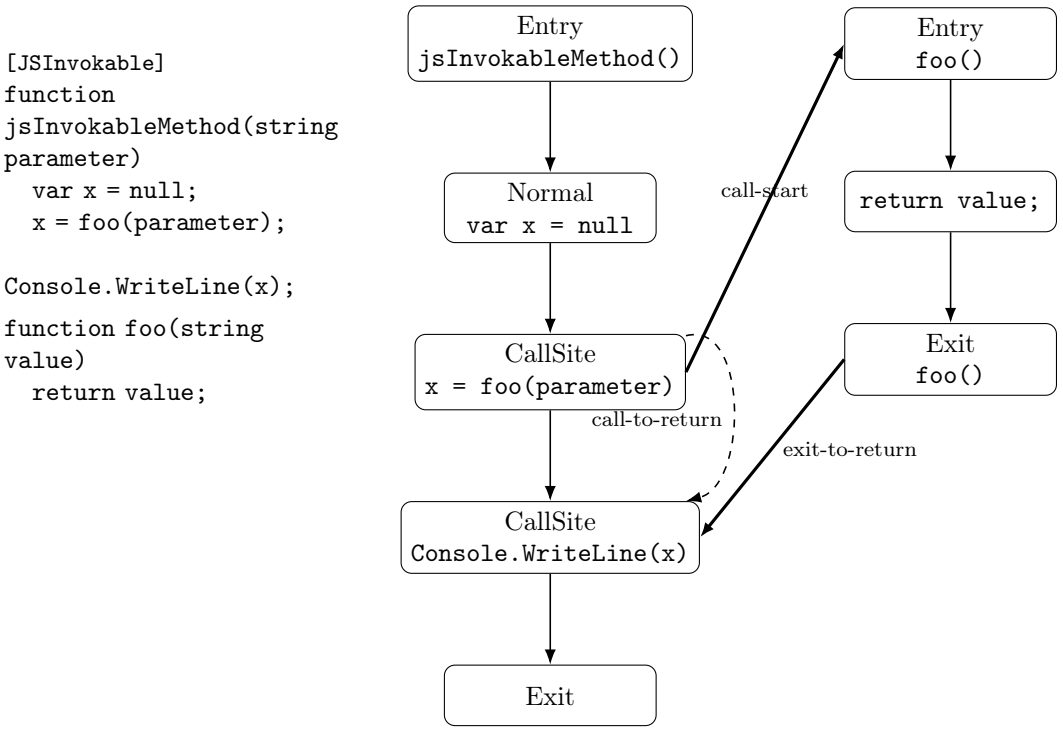

FIGURE 2.5: Example program (left) and the reachable part of its IFDS supergraph (right).

within the code is defined explicitly in the graph by two different nodes: a *call node* and a *return-site node* [13].

The supergraph G^* includes edges representing control flow within a single procedure (intraprocedural edges) and control flow between procedures (interprocedural edges). When the flow reaches a procedure call represented by a call node and its corresponding return-site node, the following edges define the control transfer [56]:

1. **Intraprocedural Call-to-Return Edge:** Connects the call node directly to the return-site node within the *same* procedure. This edge allows the propagation of dataflow facts that are local to the caller and are not affected by the execution of the called method.

2. **Interprocedural Call-to-Start Edge:** Connects the call node in the caller to the unique start node (s_p) of the *called* procedure. This edge models the transfer of control into the callee and is used to pass relevant dataflow facts (like argument states) into the callee's context.

3. **Interprocedural Exit-to-Return Edge:** Connects the unique exit node (e_p) of the *called* procedure back to the return-site node in the *caller*. This edge models the return of control and is used to pass relevant dataflow facts (like the effect on arguments or the return value state) back to the caller's context.

Figure 2.5 shows a concrete call that contains the different types of edges and the portion of the supergraph that becomes reachable from an example entry point.

A fundamental requirement for applying the IFDS algorithm is that the domain of dataflow facts must be the **powerset of a finite set D** [13, 55]. Although this condition might initially seem abstract, it can be summarized by the following key statements:

- **Finite Set D:** This is the set of all possible *individual* dataflow facts that the analysis tracks. The number of these base facts must be finite. For example, in a simple taint analysis, D might be the set of all facts of the form `tainted(v)` for every variable `v` in the program. In a type analysis, D could be the set of pairs `(v, T)` where `v` is a variable and `T` is a possible class type [55].
- **Powerset $\mathcal{P}(D)$:** The actual state computed by the analysis at any program point is a *subset* of D. The powerset $\mathcal{P}(D)$ is the set of all possible subsets of D. This means the state might be $\emptyset$, {tainted(x)}, {tainted(x), tainted(y)}, etc.
- **Set Union ($\cup$) as Merge Operator:** When control flow paths merge (e.g., after an `if` statement), the dataflow information from the different paths is combined using set union [55]. For example, if one path results in the state {tainted(x)} and another results in {tainted(y)}, the state after the merge is {tainted(x)} $\cup$ {tainted(y)} = {tainted(x), tainted(y)}.

Additionally, the flow functions associated with program statements must be **distributive** over the set union operator [13, 55]. This property ($f(A \cup B) = f(A) \cup f(B)$) is key to the algorithm's efficiency, as it allows the analysis to compute the effect of a function on a set of facts by considering its effect on each fact individually [55].

2.6.3 Extensions of IFDS

Since the introduction of the IFDS framework, it has been widely used and extended in various studies to improve its applicability, precision, or performance. Notable extensions include enhancing return-flow precision by providing caller-side context [55], concurrent execution models [57], and handling for programs in Static Single Assignment (SSA) form [55].

One practical extension proposed by Naeem et al. [55] is the *on-demand construction of the supergraph*. The goal of this approach is that, instead of requiring the entire exploded supergraph upfront, the algorithm computes only the supergraph parts reached during the analysis. This makes the algorithm more applicable to theoretically large programs, where constructing the full supergraph could be infeasible [55].

This adaptation addresses a key limitation of the original formulation. The original algorithm requires the complete exploded supergraph $G^* = (N^*, E^*)$ as input [13]. The number of nodes in this graph is roughly $|Inst| \times (|D| + 1)$, where $|Inst|$ is the number of program instructions and $|D|$ is the size of the finite set of dataflow facts [55]. When the analysis includes objects or pointers like alias or type analyses, the set D can become very large, potentially leading to billions of nodes in the theoretical supergraph. Creating and storing such a graph is often impractical due to time and memory constraints [55]. However, empirical evidence shows that only a fraction of these supergraph nodes are reachable along valid program paths from the entry points [55]. The on-demand extension uses this observation by creating the outgoing edges for a supergraph node only when encountered during the analysis, exploring only the reachable portion, and making analyses with large fact domains more manageable [55].

Several established frameworks, such as Soot [58] and WALA [59], provide implementations of the IFDS framework and are widely used for interprocedural dataflow analysis in Java and JVM-based languages. These frameworks offer mature infrastructure for working with bytecode-level representations, including control-flow graph construction and dataflow analysis, making them well-suited for IFDS applications

in the Java ecosystem. However, they are not directly applicable to the context of this study, which focuses on C# source code within .NET Maui Blazor applications. At the time of this study, there is a lack of mature IFDS-based analysis frameworks explicitly designed for the .NET ecosystem that support source-level analysis for developers. This gap motivated the development of a custom IFDS implementation for C#, addressing the unique characteristics of .NET MAUI Blazor projects and enabling static analysis at the source code level.

Table 2.1: MASVS Control Points

ID	Control
MASVS-STORAGE-1	The app securely stores sensitive data.
MASVS-STORAGE-2	The app prevents leakage of sensitive data.
MASVS-CRYPTO-1	The app employs current strong cryptography and uses it according to industry best practices.
MASVS-CRYPTO-2	The app performs key management according to industry best practices.
MASVS-AUTH-1	The app uses secure authentication and authorization protocols and follows the relevant best practices.
MASVS-AUTH-2	The app performs local authentication securely according to the platform's best practices.
MASVS-AUTH-3	The app secures sensitive operations with additional authentication.
MASVS-NETWORK-1	The app secures all network traffic according to the current best practices.
MASVS-NETWORK-2	The app performs identity pinning for all remote endpoints under the developer's control.
MASVS-PLATFORM-1	The app uses IPC mechanisms securely.
MASVS-PLATFORM-2	The app uses WebViews securely.
MASVS-PLATFORM-3	The app uses the user interface securely.
MASVS-CODE-1	The app requires an up-to-date platform version.
MASVS-CODE-2	The app has a mechanism for enforcing app updates.
MASVS-CODE-3	The app only uses software components without known vulnerabilities.
MASVS-CODE-4	The app validates and sanitizes all untrusted inputs.
MASVS-RESILIENCE-1	The app validates the integrity of the platform.
MASVS-RESILIENCE-2	The app implements anti-tampering mechanisms.
MASVS-RESILIENCE-3	The app implements anti-static analysis mechanisms.
MASVS-RESILIENCE-4	The app implements anti-dynamic analysis techniques.
MASVS-PRIVACY-1	The app minimizes access to sensitive data and resources.
MASVS-PRIVACY-2	The app prevents identification of the user.
MASVS-PRIVACY-3	The app is transparent about data collection and usage.
MASVS-PRIVACY-4	The app offers user control over their data.

Chapter 3

Previous Works

This chapter presents a review of previous works foundational to this thesis. It examines key studies on the static analysis of hybrid mobile applications, focusing on identified challenges, proposed techniques, and the current state-of-the-art. This review positions this thesis's contributions within the existing research landscape, specifically highlighting the need for specialized analysis of .NET-based hybrid frameworks like .NET MAUI Blazor.

3.1 Hybrid Mobile App Analysis

Modern mobile development often uses hybrid frameworks to avoid building separate native apps for platforms like Android and iOS [10, 11, 60]. Frameworks like React Native [11, 61], Cordova [60], and others let developers use web technologies like JavaScript or shared codebases, while still accessing native device features [10, 60]. However, mixing web and native code creates unique security risks and makes static analysis more difficult [10, 11, 60, 61]. While much research has focused on JavaScript-based hybrid frameworks, there has been less exploration of .NET-based hybrid architectures like .NET MAUI Blazor.

3.1.1 Key Challenges in Hybrid App Analysis

A significant difficulty is that hybrid apps use multiple programming languages, such as JavaScript alongside native code like Java or Kotlin [10, 11, 61]. Static analysis needs to understand both types of code and how they interact [10, 60].

The connection, or "bridge," between the web/JavaScript part and the native part is complex and a common source of security issues [10, 11, 60, 61]. Different frameworks use different bridge mechanisms. Cordova uses a specific function to call native plugins [60], while Android's WebView can expose Java objects to JavaScript [10]. React Native uses its bridge or a newer system called JSI [61]. Understanding how data and control flow across these bridges, including how data types are converted and transferred, is key to performing a thorough security assessment [10, 60]. An attacker who exploits a WebView vulnerability, such as Cross-Site Scripting, might then attempt to leverage the bridge to escalate privileges or execute unauthorized native device functions [60].

Hybrid applications also run within the context of mobile operating systems like Android, which feature component lifecycles and rely on event-driven callbacks [12, 62]. This complexity is compounded for .NET MAUI Blazor applications: the .NET MAUI framework has its own application and page lifecycle events. Blazor components have a distinct set of lifecycle methods, which all operate within the Android component

lifecycle. Static analysis tools must model these lifecycles and callback mechanisms to correctly resolve control flow and data propagation paths. Failure to do so can lead to missed behaviors (unsoundness) or incorrect findings [12, 62]. Research indicates that static analysis tools frequently fail to capture significant portions of code executed at runtime, primarily due to incomplete modeling of framework interactions and application entry points [12].

3.1.2 Static Analysis Approaches

Researchers have developed various static analysis techniques to address the complexities of hybrid and Android applications. Taint Analysis is a prevalent approach for detecting information flow vulnerabilities, tracking sensitive data (e.g., credentials, location information) from its entry points ("sources") to locations where it might be unsafely used or exposed ("sinks") [10, 62]. FlowDroid is a notable example of a static taint analysis tool for Android applications. It is recognized for its precision achieved through careful modeling of the Android lifecycle and the application of IFDS for context, flow, field, and object-sensitive analysis [62].

To handle multiple languages, tools like HybriDroid analyze Java and JavaScript, focusing on their interaction points [10]. Brucker and Herzberg developed ways to model Cordova's bridge specifically [60]. For React Native, the REUNIFY tool converts the special Hermes JavaScript bytecode into a format (Jimple) that existing Java analysis tools (like Soot, which FlowDroid uses) can understand [61]. This helps create a more complete analysis covering both the JavaScript and native parts [61].

General security testing guides, like the OWASP MASTG, provide methodologies, often focusing on native app risks [11]. However, studies like Juhola's point out that hybrid apps like React Native have additional framework-specific risks that also need testing, requiring a combination of native and web/JavaScript analysis techniques [11]. Similarly, .NET Maui Blazor applications are hybrid, which motivates this study's aim to create a SAST tool to analyze the application code developed in this framework.

A recent concern is the trade-off between how precise a static analysis is (avoiding false alarms) and how sound it is (not missing any real behavior) [12]. Studies suggest that some precise analysis techniques might miss more real runtime behavior, which is risky for security analysis [12]. Improving soundness by better understanding how apps interact with frameworks is a critical research direction [12].

3.2 Relevance to This Thesis

Although the specific languages differ (C#, Blazor, WebAssembly, JS interop versus Java/Kotlin/JavaScript), the challenges and techniques discussed in these previous works are still relevant to analyzing .NET Maui Blazor Hybrid applications. One important takeaway is the necessity of accurately modeling the specific interoperability mechanism, or "bridge", employed by .NET MAUI Blazor. Furthermore, the mixed lifecycles spanning the .NET MAUI application, Blazor components, and the underlying Android operating system are essential to correctly identifying entry points.

Building on these considerations, established techniques like taint analysis must be adapted. In .NET MAUI Blazor, the analysis must track data flows across the C#-to-JavaScript boundary, considering how data is represented and potentially transformed during interoperability. The study should also examine security-related configurations

in .NET project files, C# code, and the Android Manifest file, since these elements can affect the application's security posture.

Finally, this thesis acknowledges the crucial trade-off between analysis precision and soundness. While the IFDS framework chosen for *FlowMauiBlazor* is known for its accuracy, the inherent complexities of achieving soundness in framework-dependent applications like .NET MAUI Blazor remain a significant consideration, guiding the design towards providing actionable and reliable feedback to developers.

This thesis, therefore, builds upon these prior concepts by developing a static analysis approach tailored to .NET MAUI Blazor Hybrid applications on Android. The aim is to address the identified gap in tooling for this technology, with a particular focus on C# source code analysis using Roslyn and the IFDS framework for taint tracking across the hybrid boundary.

Chapter 4

Methodology

This study employs a multi-phase methodology to develop and evaluate a novel static analysis tool for identifying security vulnerabilities. It begins with creating a controlled testbed, the intentionally vulnerable application IVMAUIB. Next, existing community tools are used to establish a baseline security assessment. The core phase focuses on designing, implementing, and evaluating FlowMauiBlazor. Finally, FlowMauiBlazor is tested on the previously developed IID Service to assess its effectiveness on a more realistic, though still prototypical, application.

The overall methodological pipeline, detailing these phases and their interconnections, is illustrated in Figure 4.1. Each phase is elaborated upon in the subsequent subsections.

4.1 Phase 1: IVMAUIB Development

IVMAUIB, an intentionally vulnerable .NET MAUI Blazor Hybrid application, was developed to test how well security tools detect vulnerabilities. The source code of this application is publicly available on GitHub[1]. The primary objective of developing this application was to replicate common vulnerabilities typically found in hybrid mobile applications and to represent vulnerabilities from different MASVS categories.

IVMAUIB, conceptually based on the *AllSafe* application [43] and the Damn Vulnerable Hybrid Mobile App (DVHMA)[2], was developed with .NET 9.0, using the standard .NET MAUI Blazor Hybrid App template in Visual Studio.

4.1.1 Program Architecture

IVMAUIB was developed with a modular architecture to facilitate the apparent separation and demonstration of different vulnerability categories. This architecture consists of application pages, each potentially focusing on specific types of security weaknesses, and a set of services with which these pages may interact. The exposed weaknesses are built directly into the pages themselves and the services.

[1] https://github.com/mileslundqvist/IVMAUIB
[2] https://github.com/logicalhacking/DVHMA

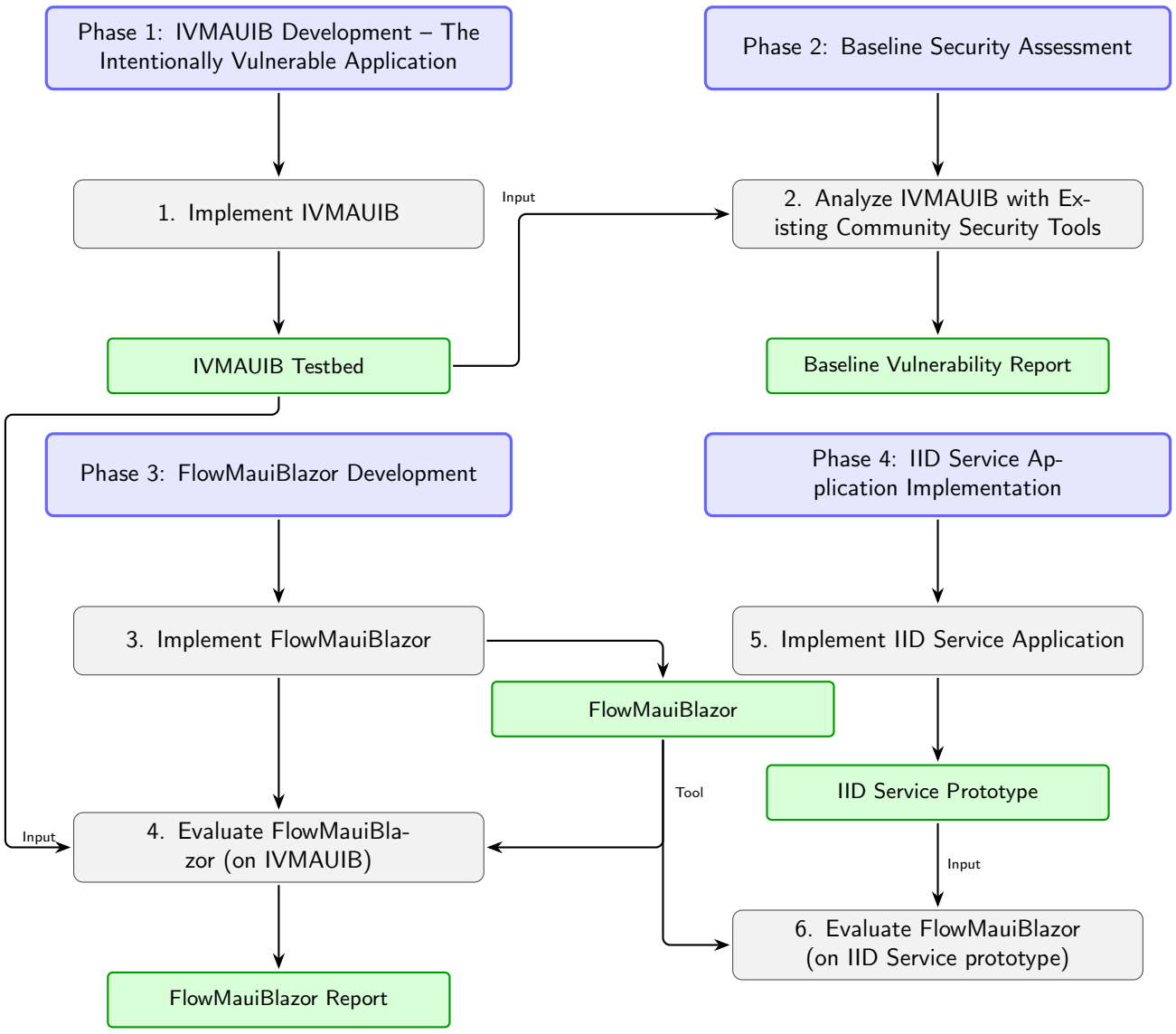

FIGURE 4.1: Pipeline illustrating key phases: IVMAUIB creation, baseline security testing, FlowMauiBlazor development and evaluation, and extension to IID Service.

Application Pages

IVMAUIB features the following distinct pages. The services utilized by each page (referenced by number corresponding to the list in the next section) are indicated in parentheses for clarity in this textual description; these dependencies are visually represented in Figure 4.2.

- **AuthVulnerabilitiesPage:** Focuses on authentication and session management vulnerabilities. (Uses Services 1, 3)
- **CodeVulnerabilitiesPage:** Demonstrates vulnerabilities arising from insecure coding practices. (Uses Service 3)
- **CryptoVulnerabilitiesPage:** Highlights weaknesses in cryptographic implementations or usage. (Uses Services 2, 3)
- **InteropVulnerabilitiesPage:** Designed to explore vulnerabilities at the boundary between different components or technologies (e.g., native code interoperation).
- **NetworkVulnerabilitiesPage:** Concerns vulnerabilities related to insecure network communication. (Uses Service 4)
- **PlatformVulnerabilitiesPage:** Focuses on platform-specific security issues (e.g., insecure use of platform APIs).
- **StorageVulnerabilitiesPage:** Deals with vulnerabilities associated with data storage. (Uses Service 3)

To support the functionalities and expose vulnerabilities, the pages can interact with the following services:

1. **InsecureAuthService:** Provides mocked authentication-related functionalities.
2. **InsecureCryptoService:** Implements cryptographic operations.
3. **InsecureDataStorageService:** Manages data persistence.
4. **InsecureNetworkService:** Simulates network interactions.

All the implemented and categorized vulnerabilities can be found in Table 5.1. The relationship between the application pages and the services they inject is depicted in Figure 4.2.

4.1.2 Preparation for Security Testing

Depending on each security-testing tool's requirements, the analysis was performed on either the unsigned, packaged APK or the project's source code.

4.2 Phase 2: Existing Security Testing Tools Assessment

The following subsections detail the specific security testing tools and methodologies applied in this study.

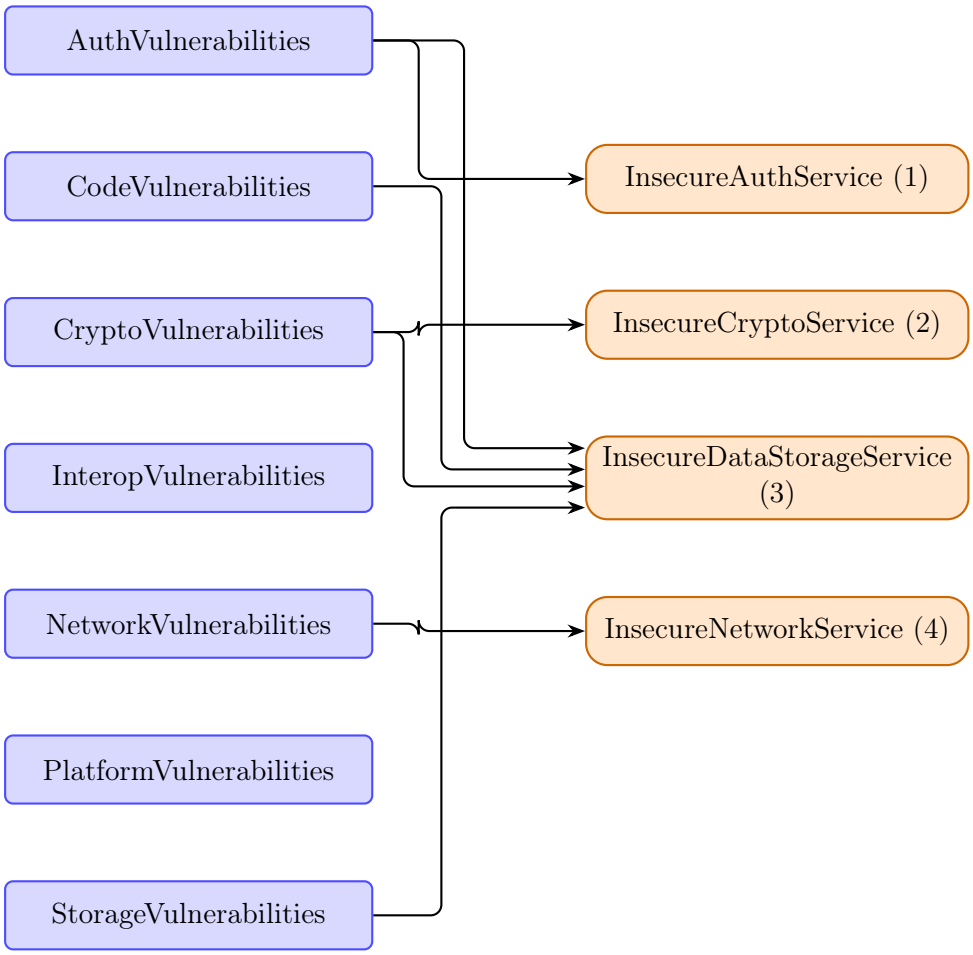

FIGURE 4.2: Architecture of IVMAUIB, showing pages and their dependencies on services.

MobSF

Mobile Security Framework (MobSF) was utilized to identify configuration issues and vulnerabilities within the developed application. MobSF was deployed via Docker using the following commands:

```
docker pull opensecurity/mobile-security-framework-mobsf:latest
docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```

Once running, MobSF takes an APK file and generates a comprehensive security report outlining the detected vulnerabilities and misconfigurations for the chosen application.

ZAP

OWASP ZAP version 2.16.0 was employed to analyze network traffic, identifying vulnerabilities exposed through HTTP(S) traffic. As ZAP is a dynamic testing tool, an Android device is needed to execute the program APK file.

For the testing environment, an Android Virtual Device (AVD) emulator was created using Pixel 5 with Android API level 33 (Android 13.0 "Tiramisu"), including Google APIs.

The following steps were executed:

1. Generated a Certificate Authority (CA) certificate via ZAP and installed it on the emulator to intercept traffic.
2. Configured the emulator to route traffic through the ZAP proxy using the commands:

   ```
   adb shell
   settings put global http_proxy 10.0.2.2:8080
   ```

   The following tests were conducted:

   - **HTTP Usage Detection:** Analyzed HTTP requests initiated by the vulnerable application, recording traffic via ZAP.
   - **SSL Certificate Validation Bypass:** Evaluated if the application accepts invalid SSL certificates by observing if HTTPS requests were completed without ZAP's CA certificate installed on the device.
   - **Credentials in Plain Text:** Monitored POST requests made by the application to identify credentials transmitted in plain text.

SonarQube

SonarQube Community Edition was leveraged for static source code analysis. The environment was configured using the official Docker image[3]. After initialization, a new .NET project was created within SonarQube. Analysis was performed using the .NET SonarScanner, providing a report highlighting source-level vulnerabilities and code quality metrics.

Xamarin Security Scanner

Xamarin Security Scanner was utilized for static analysis specific to Xamarin/.NET MAUI environments. The testing process was executed using Docker:

```
git clone <project_url>
cd xamarin-security-scanner
docker build ./XamarinSecurityScanner -t xamarin-security-scanner
docker run -v <absolute_path_to_project>:/project xamarin-security-scanner
```

The scanner provided insights into security vulnerabilities specific to Xamarin components within the application.

Security Code Scan

Security Code Scan (SCS) was integrated into the development environment via the NuGet package (version 5.6.7, SecurityCodeScan.VS1029). This tool conducted static analysis to identify common security issues in .NET code, highlighting vulnerabilities directly within the IDE.

[3] https://hub.docker.com/_/sonarqube/

4.3 Phase 3: FlowMauiBlazor Development

This section details the design and implementation of FlowMauiBlazor, the prototype SAST tool developed for this thesis using the IFDS solver algorithm. Its source code is publicly available on GitHub[4]. FlowMauiBlazor aims to perform static taint analysis on C# source code, specifically targeting .NET MAUI Blazor hybrid applications to identify potential security vulnerabilities originating from untrusted data propagation.

4.3.1 Program Architecture

FlowMauiBlazor is implemented as a standalone command-line tool in C#. It accepts either a C# project file (`.csproj`) or a solution file (`.sln`) as input. The tool leverages the Microsoft .NET Compiler Platform Roslyn [14] for static analysis capabilities, including parsing the source code, constructing and querying semantic models, and generating control-flow graphs for the methods under analysis.

The core analysis performed by the tool is taint tracking. It identifies sources of untrusted data (taint sources) and follows their propagation through the program's execution paths. The objective is to detect if tainted data reaches sensitive operations (sinks) that could cause harm, such as insecure cross-environment scripting in the Blazor context. To achieve this, the solver implements the principles of the IFDS framework for performing context-sensitive, flow-sensitive interprocedural dataflow analysis. A key feature for scalability is its demand-driven construction of the Interprocedural Control-Flow Graph (ICFG).

The architecture and interaction between the primary components of FlowMauiBlazor, highlighting the data flow from inputs through to diagnostics reporting, are illustrated in Figure 4.3.

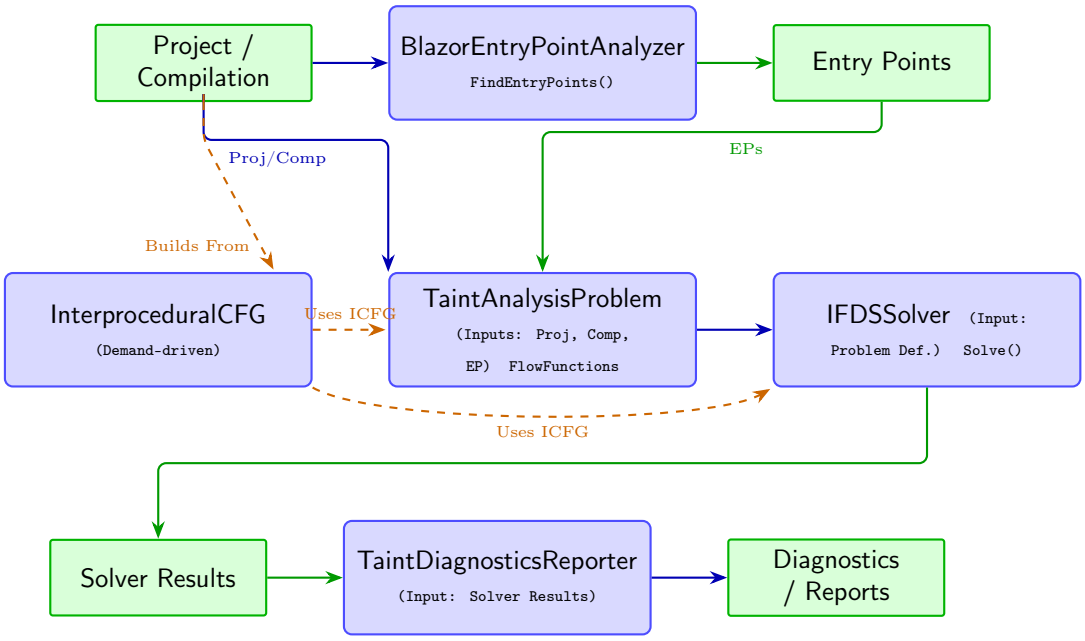

FIGURE 4.3: Architecture of FlowMauiBlazor, illustrating the flow of data and interaction between core components.

[4] https://github.com/mileslundqvist/flow-maui-blazor-analyzer

4.3.2 Demand-Driven Interprocedural Control Flow Graph Construction

Instead of constructing a complete ICFG for the entire application upfront, FlowMauiBlazor builds it incrementally as needed by the analysis. This demand-driven approach significantly improves scalability, especially for large codebases.

The process begins by identifying the initial entry points of the analysis within the target application (details on entry point identification are provided in subsection 4.3.6). Starting from these entry points, the solver utilizes Roslyn's `ControlFlowGraph` class[5] to generate precise intraprocedural control-flow graphs for individual methods on-the-fly.

Each Roslyn `ControlFlowGraph` is composed of a set of `BasicBlock` instances, representing sequences of operations without branches, except potentially at the end. Roslyn's `IOperation` objects represent the instructions within each block. For the IFDS solver, these program points are encapsulated within a custom `ICFGNode` representation. Each `ICFGNode` contains the relevant `IOperation` as the specific program point, along with contextual information derived from Roslyn, such as the containing method's symbol and the kind of operation it represents (e.g., normal statement, call-site, method entry, method exit, return-site node). This structured representation allows the solver to easily get information about the program point it is currently analyzing.

When the analysis encounters a method call-site during its traversal, it uses Roslyn's semantic model to resolve the target callee method(s). If the callee has not been analyzed yet in the current context, the solver generates the intraprocedural CFG for the callee *on demand*. This process builds the ICFG incrementally, exploring only those methods and control paths relevant to the propagation of the tracked taint facts. This avoids the significant overhead associated with pre-computing a whole-program ICFG.

4.3.3 IFDS Solver Implementation Details

The core solver adheres to the IFDS framework [13].

4.3.3.1 Fact Domain

The dataflow fact domain (D) represents the taint status of program elements. A fact can be one of the following:

- **Taint Fact:** Represents that a specific program variable (local variable, parameter, or field) holds tainted data, denoted conceptually as `tainted(x)`.
- **Zero Fact (0):** The fact representing the empty set of taints, typically holding at the beginning of the analysis entry points before any sources are encountered.

The state at any program point is a subset of D, representing all taint facts holding at that location.

[5] https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.flowanalysis.controlflowgraph?view=roslyn-dotnet-4.9.0

4.3.3.2 Flow Functions

Flow functions model how the set of taint facts transforms across program statements and control-flow edges. The solver implements flow functions to handle various scenarios precisely:

- **Intraprocedural Propagation:** Modeling assignments (e.g., `y = x`), expressions, and control structures to propagate taint facts correctly within a method. If `tainted(x)` holds before `y = x`, then `tainted(y)` and `tainted(x)` holds after.
- **Taint Sources:** Introducing new taint facts into the state. For example, at a recognized source point (like input from JavaScript), the corresponding variable fact is added to the current set.
- **Taint Sinks/Sanitization:** Modeling points where taint should be checked (sinks) or potentially removed (sanitization logic).
- **Interprocedural Flow:** Handling method calls and returns according to the IFDS edge semantics (Call-to-Start, Exit-to-Return, Call-to-Return) to map taint facts between caller and callee contexts, including parameter passing and return value handling.

4.3.3.3 Worklist Algorithm and Exploded Graph Representation

The solver uses a standard worklist algorithm described in the IFDS literature [13, 55]. The algorithm explores the reachable states of the *supergraph*. In this implementation, nodes of this graph are represented as tuples: `(ICFGNode, IFact)`. These tuples, representing reachable states, are added to and processed by the worklist. The algorithm iteratively processes these tuples from the worklist, computes the effects using the corresponding flow functions based on the `ICFGNode`'s type and operation, generates new tuples for successor states, and adds them back to the worklist. This continues until the worklist is empty, indicating that a fixed point has been reached and all reachable tainted states have been discovered. The final result comprises the set of all `TaintFact`s found to be reachable for each `ICFGNode`.

4.3.4 Handling .NET MAUI Blazor Specifics

To effectively analyze .NET MAUI Blazor hybrid applications and address the limitations observed in generic tools, FlowMauiBlazor incorporates domain-specific knowledge about the framework's architecture, particularly how C# and JavaScript interact. This tailored approach is crucial because, as identified in studies of other hybrid frameworks[10, 11], generic analyzers often lack the semantic understanding of framework-specific APIs or bridge mechanisms, leading to incomplete or inaccurate findings. FlowMauiBlazor's design focuses on precise modeling of these MAUI Blazor specifics:

- **Sources:** Methods in C# that are decorated with attributes making them invokable from JavaScript (e.g., `[JSInvokable]`) are treated as analysis entry points. Crucially, their parameters are automatically considered tainted, modeling the assumption that any data originating from the less trusted JavaScript environment should be treated as potentially unsafe.
- **Sinks:** Operations that involve sending data from C# back to the JavaScript environment (e.g., via `IJSRuntime.InvokeAsync`) are modeled as sensitive sinks.

The analysis flags instances where tainted data reaches these invocation points without appropriate validation or sanitization.

This framework-specific modeling allows the solver to track taint across the C#-JavaScript boundary, identifying potential vulnerabilities unique to Blazor hybrid applications.

4.3.5 Tool Design and User Interface

FlowMauiBlazor is designed as a command-line application for easy integration into development workflows. It requires the path to a `.csproj` or `.sln` file. By utilizing Roslyn's `Compilation` object, FlowMauiBlazor gains access to semantic information, including symbol resolution (types, methods) and control-flow structures, to enable analysis.

Upon completion, the analysis results are printed to the console. The output focuses on reporting identified taint flows that start at a defined source and terminate at a defined sink.

4.3.6 Entry Point Modelling

A fundamental prerequisite for conducting effective static data flow analysis, such as the IFDS algorithm, is precisely identifying program entry points [13, 62]. These entry points represent the locations where external data or user-initiated events first interact with the application's code, potentially introducing tainted data that needs to be tracked. In the context of .NET MAUI Blazor Hybrid applications, identifying these points presents unique challenges due to the combination of native UI elements, web UI rendering via Blazor within a WebView, and the interplay between declarative Razor syntax and imperative C# code.

This study uses Roslyn to model and identify the entry point with the Blazor portion of the hybrid application. A key aspect of this methodology is the analysis of the C# code generated by the Blazor compiler (*.razor.g.cs* files), which transforms Razor components into standard C# classes. This allows the application of Roslyn's semantic and syntactic analysis capabilities directly to the executable logic of the components, rather than attempting to parse the Razor syntax itself. The analysis is filtered to examine only types defined within the primary source assembly of the compiled project, ensuring only looking at user-written code and excluding framework or external library internals unless explicitly part of the project's implementation, for example, the components' lifecycle method overrides.

Based on the Blazor component model and MAUI Blazor Hybrid architecture, the following categories of code constructs were modelled and identified as entry points:

1. **Parameter Setters:** Blazor components receive data from parent components or the routing system via properties decorated with `[Parameter]` or `[CascadingParameter]` attributes. When such data is passed (e.g., `<ChildComponent Value="@DataFromParent" />` or via a route like `@page "/details/{Id}"`), the Blazor framework invokes the property's C# setter method (`set_Value(...)`). This setter is identified as an entry point because it's where data originating externally to the component (from parents, the URL, or cascaded values) first enters its code. The implementation uses Roslyn's semantic model to find properties with the relevant attributes and retrieves the corresponding symbol for the setter.

2. **JavaScript Invokable Methods:** The communication boundary between JavaScript running within the WebView and the .NET code is a critical entry point. C# methods decorated with the `[JSInvokable]` attribute can be directly called from JavaScript. Any data passed as arguments to these methods originates from the JavaScript environment, which could include user input, browser API results, or data manipulated by potentially untrusted scripts. Therefore, the `[JSInvokable]` method is identified as an entry point, and its parameters are considered the direct sources of potentially tainted data.

3. **Component Lifecycle Methods:** Blazor components have several lifecycle methods invoked automatically by the framework at specific points. While these methods primarily represent entry points for control flow, they are often used to perform actions like fetching data from external APIs or databases. The analysis identifies overrides or implementations of these standard lifecycle methods within the source assembly's components.

4. **Event Handler Methods:** User interactions with HTML elements within the Blazor WebView (e.g., button clicks via `@onclick`, input changes via `@onchange`) trigger C# methods defined in the component. These methods handle the application's response to user actions. The analysis identifies these handler methods by examining the compiled output of Razor components. Within this compiled representation, the UI structure and its associated logic are defined through framework-specific constructs.

   The static analysis traverses the syntax tree corresponding to the component's rendering logic to locate these event handlers. It specifically looks for patterns where UI event attributes (such as those representing clicks or input changes) are programmatically associated with C# methods. This involves identifying framework calls responsible for registering these event bindings. When such a pattern is detected, the C# method designated to handle the UI event is extracted and identified as an event handler entry point.

The collection of entry points objects generated by each containing the type, location, and relevant Roslyn symbol (`ISymbol`) for the identified entry point serves as the foundational input for the subsequent IFDS taint analysis phase. This modelling ensures that the analysis begins from relevant points where external influences can affect the application's data flow within the .NET MAUI Blazor Hybrid architecture.

4.4 Phase 4: IID Service Application Implementation

As an exploration within this thesis project, a proof-of-concept mobile application for remote IID servicing was developed. This application was built as an Android-based hybrid application using the .NET MAUI Blazor framework. The primary goal of this prototype was to investigate the feasibility of enabling users to perform self-service tasks on their IIDs, such as data log extraction and remote configuration updates, thereby addressing logistical challenges associated with traditional service center visits.

The IID service application was designed to communicate with an IID via a USB connection, typically mediated by a dongle. Key functionalities included:

- **Device Connection:** Establishing a serial communication link with the IID through the Android device's USB port. This leveraged the UsbSerialForAndroid library[6], a driver library enabling Xamarin Android (and by extension, .NET MAUI) to interface with common USB serial hardware.
- **Data Log Extraction:** Implementing logic to request and retrieve stored data logs from the IID.
- **Remote Configuration:** Providing a user interface to modify IID settings and transmitting the corresponding commands to the device to update its state.

While developing the IID service application was promising in exploring usability and operational flexibility for remote servicing, it also highlighted potential security risks inherent in such a system. Particularly concerning data exposure from a compromised smartphone and the integrity of commands sent to the IID. The process of building this hybrid application revealed the broader challenge of difficulties of existing security analysis tools in analyzing the complex interactions within .NET MAUI Blazor Hybrid applications. This observation led to the subsequent shift in the thesis's focus towards evaluating and improving static analysis techniques for this hybrid framework. This led to the development of IVMAUIB and FlowMauiBlazor. Consequently, while the IID service application served as a valuable preliminary investigation, its formal security evaluation was not pursued in the same depth as IVMAUIB, which became the central benchmark for tool assessments.

4.5 Evaluation Metrics

To assess the effectiveness of each security testing tool in detecting vulnerabilities within the application, standard ground-truth-based evaluation metrics are employed: *True Positives (TP)*, *False Positives (FP)*, and *False Negatives (FN)*. These labels are determined by comparing each tool's findings against the intentionally implemented vulnerabilities described in section 5.1. A TP indicates a correctly identified vulnerability, an FP reflects a tool warning that does not correspond to an actual vulnerability, and an FN denotes a missed detection.

These foundational metrics enable the calculation of Precision and Recall, two measures used in vulnerability detection and static/dynamic analysis benchmarking [63, 64]. Precision measures the proportion of reported findings that are correct, defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

High precision implies that the tool's alerts are trustworthy and require less manual verification. Conversely, Recall quantifies the tool's completeness in identifying known vulnerabilities, defined as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

High recall means the tool can detect most of the actual vulnerabilities in the system under analysis.

[6] https://github.com/anotherlab/UsbSerialForAndroid

Both metrics are critical from a security engineering perspective: recall reduces the risk of undetected security flaws, while precision minimizes the overhead of investigating false alerts [65]. Since security testing tools often exhibit a trade-off between precision and recall—where more aggressive detection strategies may identify additional vulnerabilities at the cost of increased false positives—this study also employs the F1 score as a balanced evaluation measure. The F1 score represents the harmonic mean of precision and recall:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score provides a single-value indicator of overall detection quality, facilitating comparison between tools with varying precision-recall characteristics [66]. The combination of TP, FP, FN counts alongside precision, recall, and F1 score metrics enables a comprehensive assessment of detection accuracy across the evaluated tools, consistent with established methodologies in software security research [63, 66].

Chapter 5

Results

This chapter presents the results of applying the methodology described in chapter 4. First, the vulnerabilities implemented in IVMAUIB are given and mapped to MASVS requirements and CWE identifiers. Then, the findings reported by each evaluated security tool are presented, including both qualitative descriptions and a summary of detections against the implemented vulnerabilities. Subsequently, a section detailing the results obtained from FlowMauiBlazor when analyzing IVMAUIB and the IID Service Application prototype is included. Finally, a comparative summary of all tools is provided against IVMAUIB.

5.1 Vulnerable Application: Implemented Vulnerabilities

IVMAUIB served as the testbed for evaluating the detection capabilities of the security tools. Table 5.1 lists the vulnerabilities intentionally implemented in the application, categorized by the corresponding MASVS requirement they violate, along with relevant CWE identifiers.

TABLE 5.1: Implemented Vulnerabilities in the Test Application Mapped to MASVS and CWE

MASVS ID	Primary CWE	Additional CWEs	Implemented Vulnerabilities (Shorthand ID: Description)
MASVS-STORAGE-1	CWE-312	CWE-922	S1: Insecure Preferences Storage
			S2: Unencrypted SQLite Database
			S3: Plaintext Files
			S5: WebView LocalStorage
MASVS-STORAGE-2	CWE-200	CWE-532, CWE-117	S4: External Storage Usage
			S7: Multiple Data Caching
			S8: Insecure Deletion
MASVS-CRYPTO-1	CWE-327	CWE-326	C1: Weak Encryption Algorithm (DES)
			C3: Static Initialization Vector
			C4: Weak Hashing Algorithm (MD5)
MASVS-CRYPTO-2	CWE-321	CWE-798	C2: Hardcoded Encryption Key
			C6: Weak Password-based Key Derivation
MASVS-AUTH-1	CWE-287	CWE-306	A1: Hardcoded Credentials
			A3: Weak Token Generation
			A5: Weak Token Validation
MASVS-AUTH-2	CWE-288	CWE-308	A2: Case-insensitive Comparison
			A6: Weak Password Policy
MASVS-AUTH-3	CWE-306	CWE-307	A8: Username Enumeration
MASVS-NETWORK-1	CWE-319	CWE-295	N1: HTTP Usage
			N2: SSL Certificate Validation Bypass
			N4: Credentials in Plain Text
MASVS-NETWORK-2	CWE-295	None	N2: SSL Certificate Validation Bypass
MASVS-PLATFORM-1	CWE-926	CWE-927	CP1: Exported Content Provider
			CP2: No Permission Checks
			CP3: Sensitive Data Exposure
MASVS-PLATFORM-2	CWE-749	CWE-79, CWE-95	P5: WebView Insecure Settings
			P6: WebView JavaScript Execution
			H1: Insecure JavaScript Interface
MASVS-PLATFORM-3	CWE-940	CWE-300	H3: Insecure Native Method Calls
			H4: DotNetObjectReference Exposure
MASVS-CODE-1	N/A	N/A	Not explicitly implemented
MASVS-CODE-2	N/A	N/A	Not explicitly implemented
MASVS-CODE-3	N/A	N/A	Not explicitly implemented
MASVS-CODE-4	CWE-20	CWE-89, CWE-94	Q1: SQL Injection
			Q2: Q2: Cross-Site Scripting (XSS)
			Q3: Insecure Deserialization
			CP4: SQL Injection in Provider
MASVS-RESILIENCE-1	CWE-353	CWE-494	R1: No Root Detection

Continued on next page

Table 5.1: Implemented Vulnerabilities in the Test Application (Continued)

MASVS ID	Primary CWE	Additional CWEs	Implemented Vulnerabilities (Shorthand ID: Description)
MASVS-RESILIENCE-2	CWE-693	CWE-354	R2: No Code Signature Verification
			R3: No Anti-Tampering Controls
MASVS-RESILIENCE-3	CWE-489	None	R5: Debug Code Enabled in Release Build
MASVS-RESILIENCE-4	CWE-391	CWE-656	R4: No Hooking Detection
MASVS-PRIVACY-1	CWE-250	N/A	P8: Excessive Permissions
MASVS-PRIVACY-2	CWE-359	CWE-200	P3: Accessing Device Identifiers
*			P4: Sensitive Device Info Access
MASVS-PRIVACY-3	N/A	N/A	Not explicitly implemented
MASVS-PRIVACY-4	N/A	N/A	Not explicitly implemented

5.2 Existing Security Tools: Findings

The following subsections detail the findings reported by each of the existing security tools when analyzing the vulnerable application described in section 5.1.

5.2.1 MobSF Findings

MobSF's static analysis of the APK reported the following potential issues:

- **CWE-312 – Cleartext Storage of Sensitive Information:**
  Triggered by the Glide media-management library. Because this artefact is introduced only in the *debug* build and not by the application's own code, the finding is treated as a false positive in this study.
- **CWE-489 – Active Debug Code:**
  Presence of debug artefacts attributable to the build configuration.
- **CWE-926 – Improper Export of Android Application Components** —
  An exported `ContentProvider` was detected, matching the intentionally introduced vulnerability CP1.
- **CWE-922 – Insecure Storage of Sensitive Information:**
  The manifest flag `android:allowBackup="true"` permits unencrypted backups that may contain sensitive data (related to MASVS-STORAGE-1).
- **CWE-250 – Execution with Unnecessary Privileges:**
  MobSF highlighted several requested permissions (e.g. network, storage) and recommended manual validation.
- **CWE-532 – Insertion of Sensitive Information into Log File:**
  Also raised through Glide and therefore considered a false positive (relevant to MASVS-STORAGE-2).

MobSF focused on manifest-level issues and artefacts introduced by the debug build or third-party libraries, detecting a few code-level weaknesses (e.g. cryptographic or authentication flaws). A cross-reference between MobSF's output and the deliberately injected MASVS vulnerabilities is provided in Table 5.2.

Table 5.2: MobSF Findings Summary by MASVS Category

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	1/7	2
MASVS-CRYPTO	5	0/5	0
MASVS-AUTH	6	0/6	0
MASVS-NETWORK	3	0/3	0
MASVS-PLATFORM	9	1/9	0
MASVS-CODE	4	0/4	0
MASVS-RESILIENCE	4	1/4	0
MASVS-PRIVACY	3	1/3	0
Total	41	4/41	2

5.2.2 ZAP Findings

OWASP ZAP, used for dynamic network analysis, identified the following issues during the manual exploration and testing phase:

- **CWE-200 / CWE-319 - Exposure of Sensitive Information / Cleartext Transmission:**
  Detected that the application permits clear-text HTTP traffic, directly identifying vulnerability N1. This was observed when the application made HTTP requests as configured.
- **CWE-295 - Improper Certificate Validation:**
  Successfully identified that the application bypasses SSL/TLS certificate validation (vulnerability N2). This was confirmed when the application communicated over HTTPS through ZAP without the ZAP CA certificate installed on the emulator.
- **CWE-319 - Cleartext Transmission - for Credentials:**
  Observed credentials sent in plain text during POST requests (vulnerability N4) over HTTP.

ZAP's dynamic analysis uncovered network-layer weaknesses, specifically plaintext communication channels and insufficient certificate validation. The detailed mapping of ZAP's findings to the intentionally injected MASVS vulnerabilities is presented in Table 5.3.

Table 5.3: ZAP Findings Summary by MASVS Category

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	0/7	0
MASVS-CRYPTO	5	0/5	0
MASVS-AUTH	6	0/6	0
MASVS-NETWORK	3	3/3	0
MASVS-PLATFORM	9	0/9	0
MASVS-CODE	4	0/4	0
MASVS-RESILIENCE	4	0/4	0
MASVS-PRIVACY	3	0/3	0
Total	41	3/41	0

5.2.3 SonarQube Findings

SonarQube Community Edition's static analysis of the source code reported the following:

- **CWE-95 – Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection'):**
  Potential use of `eval()` in `index.html`, relevant to WebView security (vulnerability P6 or H1, depending on execution context).
- **CWE-200 / CWE-319 – Cleartext Communication:**
  Configuration patterns that allow clear-text HTTP traffic, matching vulnerability N1.

- **CWE-489 – Debug Code Enabled:**
  Manifest entry `android:debuggable="true"`.
- **CWE-922 – Insecure Backup Configuration:**
  Manifest entry `android:allowBackup="true"`.
- **CWE-327 – Use of Weak Hash (MD5):**
  Occurrence of the MD5 hashing algorithm, mapping to vulnerability C4.
- **CWE-327 – Use of a Broken or Risky Cryptographic Algorithm (DES):**
  Use of the DES cipher, mapping to vulnerability C1.
- **CWE-250 – Execution with Unnecessary Privileges:**
  Requested permissions flagged for review.
- **CWE-926 – Improper Export of Android Application Components:**
  Exported `ContentProvider` corresponding to vulnerability CP1.
- **CWE-295 – Improper Certificate Validation:**
  Code that disables SSL/TLS certificate validation, corresponding to vulnerability N2.

The findings span manifest configuration, cryptographic algorithm usage, network transport settings, and dynamic code evaluation. A category-level summary of TP and FP detections is provided in Table 5.4.

Table 5.4: SonarQube Findings Summary by MASVS Category

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	1/7	0
MASVS-CRYPTO	5	2/5	0
MASVS-AUTH	6	0/6	0
MASVS-NETWORK	3	2/3	0
MASVS-PLATFORM	9	2/9	0
MASVS-CODE	4	0/4	0
MASVS-RESILIENCE	4	1/4	0
MASVS-PRIVACY	3	1/3	0
Total	41	9/41	0

5.2.4 Xamarin Security Scanner Findings

Static analysis with the Xamarin Security Scanner produced the following findings:

- **CWE-922 – Insecure Backup Configuration:**
  Manifest entry `android:allowBackup="true"`.
- **CWE-489 – Debug Code Enabled:**
  Manifest entry `android:debuggable="true"`.

These findings concern common security misconfigurations detected in the project's manifest and build settings. A MASVS-aligned summary of TP and FP counts is provided in Table 5.5.

Table 5.5: Xamarin Security Scanner Findings Summary by MASVS Category

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	1/7	0
MASVS-CRYPTO	5	0/5	0
MASVS-AUTH	6	0/6	0
MASVS-NETWORK	3	0/3	0
MASVS-PLATFORM	9	0/9	0
MASVS-CODE	4	0/4	0
MASVS-RESILIENCE	4	1/4	0
MASVS-PRIVACY	3	0/3	0
Total	41	2/41	0

5.2.5 Security Code Scan Findings

Security Code Scan (SCS), integrated via NuGet, reported the following vulnerabilities directly from the C# code analysis during the build process:

- **CWE-295 – Improper Certificate Validation:**
  Code that disables SSL/TLS certificate validation, mapping to vulnerability N2.
- **CWE-327 / CWE-328 – Use of Weak Hash Function (MD5):**
  The MD5 hashing algorithm is invoked, mapping to vulnerability C4.
- **CWE-327 – Use of a Broken or Risky Cryptographic Algorithm (DES):**
  Use of the DES cipher, mapping to vulnerability C1.

The findings originate from static C# codebase analysis and relate primarily to cryptographic algorithm strength and certificate-validation logic. A MASVS-based summary of TP and FP counts is provided in Table 5.6.

Table 5.6: Security Code Scan Findings Summary by MASVS Category

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	0/7	0
MASVS-CRYPTO	5	2/5	0
MASVS-AUTH	6	0/6	0
MASVS-NETWORK	3	1/3	0
MASVS-PLATFORM	9	0/9	0
MASVS-CODE	4	0/4	0
MASVS-RESILIENCE	4	0/4	0
MASVS-PRIVACY	3	0/3	0
Total	41	3/41	0

5.3 FlowMauiBlazor Findings (on IVMAUIB)

FlowMauiBlazor, the Roslyn-based static analyzer developed for this thesis, performs intra- and interprocedural taint analysis. It reports data-flow instances where a tainted value reaches a defined sink.

FlowMauiBlazor generated **14** distinct taint-flow reports when analyzing IVMAUIB. A summary of the detected taint flows, categorized by the identified vulnerability type, is presented below. Detailed reports for each of the 14 findings, including specific source/sink locations and propagation traces, are provided in Appendix A.

- **Hardcoded Encryption Key (C2):** Tainted data from encryption with a hardcoded key flowed into a `Console.WriteLine` sink (Finding #1).
- **Weak Password-based Key Derivation (C6):** Tainted data derived from a password using a weak hashing method flowed into a `Console.WriteLine` sink (Finding #2).
- **Insecure JS Execution/Interface (P6/H1):** Tainted JavaScript code flowed into the `JSRuntime.InvokeAsync<string>("eval", ...)` sink (Finding #3).
- **Insecure Native Method Calls (H3):** Tainted command string received from JavaScript flowed into a `Console.WriteLine` sink (Finding #4).
- **Insecure Preferences Storage (S1):** Tainted data (username, password, type) received from JavaScript flowed into `Preferences.Default.Set` sinks (Findings #5, #6, #7, #8, #11, #14).
- **Plaintext Files (S3):** Tainted authentication token flowed into a `File.WriteAllText` sink (Finding #12).
- **Multiple Data Caching (S7):** Tainted data (username, type) received from JavaScript or generated locally flowed into `Console.WriteLine` sinks (Findings #9, #10, #13).

The seven unique detected vulnerabilities to MASVS categories are mapped in Table 5.7.

Table 5.7: FlowMauiBlazor Findings Summary by MASVS Category (on IVMAUIB)

MASVS Category	Implemented Vulns (Count)	TP Count	FP Count
MASVS-STORAGE	7	3	0
MASVS-CRYPTO	5	2	0
MASVS-AUTH	6	0	0
MASVS-NETWORK	3	0	0
MASVS-PLATFORM	9	2	0
MASVS-CODE	4	0	0
MASVS-RESILIENCE	4	0	0
MASVS-PRIVACY	3	0	0
Total	**41**	**7**	**0**

5.4 Analysis of the IID Service Application with FlowMauiBlazor

In addition to IVMAUIB, FlowMauiBlazor was also run on the IID Service Application prototype described in section 4.4. This application was developed as an initial exploration of remote IID servicing and was not designed with the same set of intentional, MASVS-mapped vulnerabilities as IVMAUIB. FlowMauiBlazor's analysis of the IID Service Application detected no taint flows matching the analyzer's currently defined source-to-sink patterns.

5.5 Summary of Findings

This section provides a quantitative comparison of the detection capabilities of the evaluated tools based on their performance against the set of implemented vulnerabilities detailed in Table 5.1. Table 5.8 summarizes these overall metrics for each tool, offering a comparative overview of their effectiveness in the context of this study.

Table 5.8: Comparison of Tools Performance Metrics

Tools	TP	FP	FN	Recall (%)	Precision (%)	F1-Score
MobSF	4	2	35	10.3	66.7	17.8
ZAP	3	0	38	7.3	100.0	13.6
SonarQube	9	0	32	22.0	100.0	36.0
Xamarin Sec. Scanner	2	0	39	4.9	100.0	9.3
Security Code Scan	3	0	38	7.3	100.0	13.6
FlowMauiBlazor	7	0	34	17.1	100.0	29.2

Chapter 6

Discussion

This chapter analyzes the results, focusing on the strengths, limitations, and implications of existing security testing tools and the FlowMauiBlazor prototype developed in this study. It aligns the experimental findings with the research goals and evaluates how effectively existing solutions tackle the security issues faced by .NET MAUI Blazor Hybrid applications. Additionally, the chapter discusses the broader implications of the methodology, the insights gained from testing prototype applications, and the technical challenges faced during tool development.

6.1 Existing Tools Detection

The evaluation of existing security testing tools against the intentionally vulnerable .NET MAUI Blazor application revealed a varied landscape of detection capabilities, as detailed in Chapter 5. A key observation, consistent with the summarized findings in Table 5.8, is that while all tested tools successfully identified at least one vulnerability, their focus and effectiveness differed significantly, leaving some gaps, particularly concerning .NET MAUI Blazor-specific vulnerabilities.

6.1.1 Dynamic Analysis

The methodologies used by the tools influence their findings. OWASP ZAP, this study's only dynamic analysis tool, excelled in identifying network-related vulnerabilities. As shown in Table 5.3, ZAP achieved perfect detection (3/3 TPs) for the MASVS-NETWORK category, identifying HTTP usage, SSL certificate validation bypass, and credentials sent in plain text. This highlights that dynamic testing for observing runtime behavior, such as network transmission to examine requests and responses, is a good approach for finding weaknesses, as also mentioned in OWASP MASTG [20].

6.1.2 Static Analysis

Among the static analysis tools, approaches also varied. MobSF performed its analysis on the packaged APK file, decompiling and examining the code as noted by Kohli et al. [41] and Chandra et al. [42]. This allowed it to find manifest issues and some library-related concerns. The library-related concerns were deemed FPs in this study's context, as the goal was to assess application code.

A notable similarity between several static analysis tools was the detection of common manifest-level Android misconfiguration. For instance, MobSF, SonarQube, and the Xamarin Security Scanner all flagged issues such as android:allowBackup="true" and

android:debuggable="true" in the build. MobSF and SonarQube also identified the exported content provider. This suggests that the chosen tools have some capability to identify well-known Android configuration weaknesses.

In contrast, SonarQube and Security Code Scan analyzed the C# source code directly. SonarQube demonstrated broader source code analysis capabilities, identifying issues like weak cryptographic algorithm usage, cleartext communication, and improper certificate validation, resulting in the highest number of TPs (9) among existing tools. SCS, specializing in .NET security, also effectively found code-level flaws like DES, MD5, and SSL validation bypass usage. The Xamarin Security Scanner, while also a source code analyzer, had a much narrower focus, primarily on manifest configurations.

Despite these successes in their respective domains, none of the existing tools could effectively identify unique or particularly vulnerable areas within the .NET MAUI Blazor Hybrid code of IVMAUIB. This underscores a gap in current tooling when addressing the security challenges posed by modern hybrid application architectures, a gap this thesis aimed to explore and begin addressing with FlowMauiBlazor.

6.1.3 Key Takeaways

The evaluation of existing security testing tools yields several important insights:

- **Methodological specialization matters**: Dynamic analysis tools (ZAP) excel at runtime behavior detection, particularly network vulnerabilities. In contrast, static analysis tools vary significantly in their scope and effectiveness depending on their analysis approach.
- **Common vulnerabilities are well-covered**: Standard Android misconfigurations and well-known security weaknesses are consistently detected across multiple tools, indicating mature detection capabilities for established vulnerability patterns.
- **Source code analysis provides deeper insights**: Tools analyzing C# source code directly (SonarQube, SCS) demonstrated superior detection rates than those analyzing compiled artifacts, particularly for code-level security issues.
- **Critical gap in hybrid application security**: None of the evaluated tools effectively addresses .NET MAUI Blazor-specific hybrid application vulnerabilities, revealing a significant limitation in current security testing capabilities for modern cross-platform development frameworks.

These findings highlight the strengths of existing tooling in traditional vulnerability detection and the need for specialized approaches to address emerging hybrid application architectures.

6.2 FlowMauiBlazor Detection (on IVMAUIB)

FlowMauiBlazor, the IFDS-based taint analysis tool developed as part of this thesis, was designed to identify vulnerabilities originating from tainted data propagation within .NET MAUI Blazor Hybrid applications, focusing on the C#-JavaScript interoperability boundary. The results presented in section 5.3 demonstrate FlowMauiBlazor's capability in this domain.

6.2.1 Detection Results Overview

FlowMauiBlazor successfully identified seven unique implemented vulnerabilities, generating 14 distinct taint flow reports, as detailed in Table 5.7. The prototype achieved zero false positives among its detections, resulting in 100% precision, although this metric should be interpreted carefully. The high precision reflects FlowMauiBlazor's conservative approach, as it only reports vulnerabilities that match its well-defined source-to-sink patterns. While this ensures reliability for the patterns it covers, the tool's 17.1% recall (7 of 41 vulnerabilities detected) indicates substantial detection gaps that would need addressing for comprehensive security analysis.

6.2.2 Vulnerability Category Analysis

The detected true positives span several MASVS categories, demonstrating the tool's effectiveness across different vulnerability types:

MASVS-STORAGE (3 TPs): FlowMauiBlazor identified vulnerabilities S1 (Insecure Preferences Storage), S3 (Plaintext Files), and S7 (Multiple Data Caching). These detections typically involved data originating from JavaScript-invokable methods flowing into sinks such as `Preferences.Default.Set`, `File.WriteAllText`, or `Console.WriteLine`.

MASVS-CRYPTO (2 TPs): The tool detected vulnerabilities C2 (Hardcoded Encryption Key) and C6 (Weak Password-based Key Derivation). These were identified when data processed by cryptographic service methods, which themselves employed hardcoded keys or weak derivation techniques, was subsequently passed to logging sinks, demonstrating the propagation of taint through cryptographic operations.

MASVS-PLATFORM (2 TPs): FlowMauiBlazor identified vulnerabilities P6 (WebView JavaScript Execution)/H1 (Insecure JavaScript Interface) and H3 (Insecure Native Method Calls). Finding #3 revealed tainted JavaScript code from user input flowing into the `JSRuntime.InvokeAsync("eval", ...)` sink, directly representing an insecure JavaScript execution vulnerability. Finding #4 demonstrated a tainted command string, originating from a `[JSInvokable]` method parameter, being logged after simulated native execution.

6.2.3 Tool Design Impact on Performance

FlowMauiBlazor's success in these areas is directly attributable to its domain-specific modeling for .NET MAUI Blazor applications. By defining parameters of `[JSInvokable]` methods as primary taint sources and methods such as `IJSRuntime.InvokeAsync` and standard MAUI Blazor I/O operations (including `Preferences.Set` and `Console.WriteLine`) as sinks, the tool was specifically tailored to detect cross-environment scripting issues and insecure data handling originating from the JavaScript context.

Compared to the other tools evaluated, FlowMauiBlazor achieved a recall of 7/41 (17.1%), which appears modest in absolute terms. However, this metric should be interpreted within the tool's specialized design focus. The tool's key strength lies in its precision and framework-specific approach to .NET MAUI Blazor applications, successfully identifying specific taint flows related to the C#-JavaScript boundary that more generic tools might overlook due to insufficient framework understanding.

6.2.4 Limitations and Scope Constraints

The current source and sink model's specialized focus also defines FlowMauiBlazor's inherent limitations. The tool did not detect vulnerabilities in categories such as MASVS-AUTH, MASVS-NETWORK, MASVS-CODE, MASVS-RESILIENCE, or certain MASVS-PRIVACY areas. This limitation is expected and by design, as the prototype was not developed with specific detection rules for these broader vulnerability classes.

6.2.5 Key Takeaways

The evaluation of FlowMauiBlazor yields several important insights regarding specialized security analysis tools:

- **Framework understanding is crucial**: FlowMauiBlazor's ability to detect C#-JavaScript boundary vulnerabilities highlights the importance of incorporating platform-specific knowledge into security analysis tools.
- **Specialized tools complement generic ones**: While achieving lower overall recall (17.1%) compared to some generic tools, FlowMauiBlazor identified unique vulnerability patterns that existing tools missed entirely.
- **Taint Analysis is effective for hybrid applications**: The IFDS-based approach successfully tracked data flow across the JavaScript-C# boundary, proving viable for complex hybrid application architectures.

These findings demonstrate the viability of targeted taint analysis for uncovering security vulnerabilities in .NET MAUI Blazor Hybrid applications, establishing FlowMauiBlazor as a promising foundation for developing more comprehensive security analysis tools for this specific technology stack.

6.3 IID Service Application Analysis and Implications

FlowMauiBlazor was also applied to the IID service application prototype. Unlike IVMAUIB, the IID application was not intentionally seeded with vulnerabilities, as it was developed during the thesis's initial exploratory phase. The analysis did not detect taint flows matching the analyzer's defined source-to-sink patterns.

These findings do not mean the IID application is free of vulnerabilities, as FlowMauiBlazor currently detects only specific taint patterns. However, the results have two important implications for this study. First, applying FlowMauiBlazor to a codebase that was intentionally seeded with vulnerabilities, unlike IVMAUIB, suggests that the tool does not generate false positives or arbitrary alarms. Second, the challenges encountered during the IID application's development, especially regarding the security of its hybrid interactions, highlighted the need for improved, framework-specific analysis tools like FlowMauiBlazor. This need directly motivated the research direction of this thesis.

While the IID service application served as a valuable secondary test case, a broader evaluation of more diverse, real-world .NET MAUI Blazor applications would strengthen the validation of FlowMauiBlazor's effectiveness and help identify further areas for improvement. However, access to such applications was not available within the scope of this thesis. Moreover, because FlowMauiBlazor is currently limited, continued development of the tool itself would be necessary in parallel with expanding testing.

6.4 FlowMauiBlazor Implementation

Implementing an IFDS-based taint analyzer specifically for .NET MAUI Blazor Hybrid applications required building foundational infrastructure that is already available in other ecosystems. Unlike Java researchers who can leverage mature frameworks like Soot [58] and WALA [59], no equivalent open-source IFDS implementation was found for .NET source-level analysis. This necessitated developing core components from scratch, including parsing C# with Roslyn, constructing the interprocedural control-flow graph, implementing IFDS flow functions, and modeling specific framework semantics, each presenting its own set of technical challenges.

6.4.1 Roslyn Integration Challenges

The Roslyn compiler is central to FlowMauiBlazor's capability to analyze C# source code, offering essential APIs for parsing syntax trees, building semantic models, and constructing intraprocedural control flow graphs. However, IFDS analysis requires different representations, for instance, an interprocedural control flow graph (ICFG). In this graph, each node represents a program point, and edges represent possible execution paths.

Bridging the gap between Roslyn's structure and the requirements of IFDS analysis demanded significant development effort. Each of Roslyn's IOperation objects, which represent individual C# statements such as assignments or method calls, needed to be transformed into custom ICFGNode objects. These nodes must capture not only the actions of the code but also how data flows through it. For instance, a simple method call in C# must be broken down into multiple ICFG nodes that represent the call site, method entry, and handling of the return value. Although this translation process is conceptually straightforward, it was proved to be complex and consumed a substantial portion of the implementation time.

6.4.2 Entry Point Modeling Complexity

One significant challenge for static analysis in .NET MAUI Blazor Hybrid applications is that it does not have a single main method that marks the start of program execution. This issue is common in event-driven, framework-based applications, as noted by Arzt et al. in their work on FlowDroid for Android [62]. In MAUI Blazor, program flow instead begins from multiple entry points invoked by the underlying .NET MAUI and Blazor frameworks in response to various events, as detailed in subsection 4.3.6.

Precisely identifying and modeling these framework-invoked entry points is fundamental for achieving a useful analysis that can capture relevant taint flows, as also presented in previous studies [12, 62]. The development of the entry point model required analysis of the .NET MAUI Blazor framework's mechanisms for initiating application behavior. This analysis ensured that the IFDS solver within FlowMauiBlazor initiates its taint analysis from relevant locations where external data or framework events first interact with the application's C# code.

The accuracy and thoroughness of this entry point model directly determine the overall precision of the taint analysis. If any true entry points are missed in this model, the analysis may fail to detect certain taint flows that originate from those points, leading to false negatives. Therefore, continuously improving and expanding this entry point model is an important area if the tool is to be further developed.

6.4.3 ICFG Construction Strategy

The construction of the interprocedural control flow graph (ICFG) was approached with scalability considerations, leading to the demand-driven strategy described in Section 4.3.2. This approach avoided the prohibitive computational cost of building a whole-program ICFG upfront, instead generating control flow information on demand as the analysis progressed.

Nevertheless, implementing this on-demand generation using Roslyn, including resolving method call sites and generating CFGs for callees, represented one of the most complex aspects of FlowMauiBlazor's architecture. The demand-driven approach required careful coordination between the IFDS solver and the ICFG construction mechanism to ensure completeness while maintaining performance.

6.4.4 IFDS Solver Implementation

The IFDS solver, based on principles outlined by Reps, Horwitz, and Sagiv [13], formed FlowMauiBlazor's analytical core. Implementation involved defining the taint fact domain, implementing flow functions for intraprocedural and interprocedural data propagation, and developing the worklist algorithm to explore the supergraph effectively.

The current prototype simplifies the definition of taint sources, sinks, and sanitizers, creating an opportunity for improvements. It matches predefined patterns stored in JSON files, like identifying methods callable from JavaScript as sources and risky methods as sinks. However, this process does not consider specifics like parameter indexes and does not differentiate between parameters within method calls. While this approach effectively targets Blazor vulnerabilities, developing a more detailed matching system that considers more metadata could enhance the precision and effectiveness of the tool.

6.4.5 Current Limitations and Soundness Considerations

The current implementation of flow functions may not comprehensively cover every C# language construct or .NET library interaction in a thoroughly sound manner. This limitation means scenarios may exist where tainted facts are not propagated correctly, potentially leading to false negatives. Achieving complete soundness across a complex language and framework represents a fundamental challenge in static analysis, often requiring iterative refinement and extensive handling of edge cases, as demonstrated in studies of other complex ecosystems like Android [12].

FlowMauiBlazor focused on core taint propagation mechanisms relevant to common .NET MAUI Blazor patterns, prioritizing practical applicability over theoretical completeness. Performance and optimization were secondary considerations. The primary objective was to demonstrate sound, precise, context-aware, and flow-sensitive taint analysis for the target hybrid framework.

6.4.6 Handling of Blazor `@bind` Variables in Taint Analysis

A specific challenge in statically analyzing Blazor applications arises from the two-way data binding mechanism, specifically the `@bind` directive. These directives establish implicit links between UI elements and component fields or properties, requiring special handling in taint analysis.

6.4.6.1 Identified Limitation

The primary difficulty stems from event-driven programming patterns and Blazor's binding implementation. Blazor invokes compiler-generated lambda functions to update corresponding C# fields/properties when users interact with bound UI elements. For example, an input bound to `this.password` might result in a hidden assignment like `this.password = __ui_value;` within such a lambda.

The IFDS analysis identifies these generated lambdas and target fields/properties as specific entry point types: `BindingCallback`. The `NormalFlow` function within the IFDS solver recognizes assignments within these lambdas and generates new `TaintFact` instances for bound fields.

The limitation arises when bound fields are subsequently used in different event handlers or methods without direct, preceding operations within the same method that would explicitly propagate taint. Consider the following Blazor component:

```
@page "/counter"

<input type="text" @bind="password" />
<button @onclick="handleClick"/>

@code {
    private string password = "";

    private void handleClick()
    {
        // SINK: 'password' is used directly here
        Console.WriteLine(password);
    }
}
```

In this scenario, no explicit data-flow edge within the ICFG directly carries the `TaintFact` for `this.password` from the binding lambda's execution state to the program point before the field read within `handleClick()`. Standard IFDS propagation relies on such paths or incoming `TaintFact` instances at points of use, potentially leading to false negatives.

6.4.6.2 Current Mitigation and Proposed Enhancement

To address this limitation, FlowMauiBlazor implements a specific mechanism within the `NormalFlow` function's taint generation process:

1. **Entry Point Identification:** The initial analysis phase identifies all fields/properties involved in `@bind` directives and records them for later access during analysis.
2. **Taint on Read:** When encountering operations that read fields or properties, the `NormalFlow` method consults the list of identified `BindingCallback` entry points. If the field/property being read matches a symbol from such an entry point, the read operation is treated as a taint source, generating a `TaintFact` for that field/property.

This approach over-approximates by treating any field subject to a `@bind` directive as tainted whenever accessed in assignments. This ensures that taint introduced by UI

binding mechanisms is available for propagation when bound variables are used, even in event handlers not directly called by binding lambdas.

6.4.7 Key Takeaways

The FlowMauiBlazor implementation process reveals several important insights about developing specialized static analysis tools:

- **Compiler integration complexity**: While platforms like Roslyn provide powerful APIs, building effective analysis layers requires substantial development effort and deep understanding of compiler internals.
- **Framework-specific modeling is crucial**: Accurate entry point identification and modeling for event-driven frameworks like .NET MAUI Blazor is both critical and challenging, directly impacting analysis soundness.
- **Scalability requires careful design**: Demand-driven ICFG construction proved essential for managing computational complexity while maintaining analysis completeness.
- **Practical trade-offs are necessary**: Balancing theoretical soundness with practical applicability requires focused implementation efforts on core vulnerability patterns rather than comprehensive language coverage.
- **Framework-specific challenges need specialized solutions**: Issues like Blazor's `@bind` mechanism require custom handling approaches that generic analysis frameworks would likely miss.

These findings demonstrate the feasibility and complexity of developing specialized static analysis tools for modern hybrid application frameworks, highlighting the importance of domain-specific knowledge in creating effective security analysis capabilities.

Chapter 7

Conclusion

This thesis investigated secure development practices for .NET MAUI Blazor Hybrid applications on Android through a comprehensive approach encompassing benchmark development, tool evaluation, and prototype implementation. The research addressed a gap in security analysis capabilities for modern hybrid mobile application frameworks by developing specialized tooling and empirically evaluating existing solutions.

7.1 Research Questions Answered

The empirical evaluation addressed the research questions outlined in Chapter 1:

RQ1: How effective are existing open-source static and dynamic security-testing tools at detecting MASVS-aligned vulnerabilities in a .NET MAUI Blazor Hybrid Android application?

Existing tools demonstrated reliable detection capabilities for well-established mobile security misconfigurations and network-related vulnerabilities. Dynamic analysis tools like OWASP ZAP excelled at runtime behavior analysis, while static analysis tools like SonarQube showed strong performance for general C# code security issues. However, all evaluated tools showed significant limitations in detecting hybrid architecture-specific vulnerabilities, achieving overall recall rates that left security gaps unaddressed. The tools' generic approaches proved insufficient for framework-specific security concerns inherent to .NET MAUI Blazor applications.

RQ2: To what extent does *FlowMauiBlazor*, a prototype IFDS taint analyzer, increase MASVS vulnerability-detection coverage compared with existing static analyzers for .NET MAUI Blazor Hybrid applications?

FlowMauiBlazor demonstrated measurable improvements in detecting framework-specific vulnerabilities, particularly those involving C#-JavaScript boundary interactions and insecure local storage patterns. The tool identifies seven unique vulnerabilities that existing tools completely missed. This specialized approach proved essential for addressing security concerns specific to hybrid application architectures, validating the hypothesis that framework-aware analysis improves security assessment capabilities.

7.2 Key Contributions

This thesis makes several contributions to the field of mobile application security, centered around two implementations:

IVMAUIB - Security Benchmark: The development of IVMAUIB (Intentionally Vulnerable .NET MAUI Blazor) represents the first, MASVS-aligned vulnerable application specifically designed for .NET MAUI Blazor security research. This benchmark application, which contains 41 documented vulnerabilities across multiple security categories, provides a standardized evaluation platform that enables a methodical assessment of security tools and techniques for hybrid mobile applications. IVMAUIB addresses a gap in the research community by providing a framework-specific testing environment for security analysis validation.

FlowMauiBlazor - Specialized Security Analysis Tool: FlowMauiBlazor demonstrates the feasibility and effectiveness of IFDS-based taint analysis tailored explicitly for hybrid mobile applications. This Roslyn-based static analysis tool successfully identifies vulnerabilities at the C#-JavaScript interoperability boundary that generic security tools overlook entirely. The prototype reported zero false positives by only flagging taint flows that match the defined source-to-sink patterns. However, its 17.1% recall shows there are still significant detection gaps with IVMAUIB. Overall, FlowMauiBlazor lays a foundation for developing more effective security analysis tools for modern hybrid application frameworks, such as .NET MAUI.

Empirical Security Tool Evaluation: The systematic assessment of five existing open-source security tools provides insights into current capabilities and limitations in hybrid application security analysis. This evaluation reveals both the strengths of existing approaches for traditional mobile security concerns and their significant gaps in addressing hybrid architecture-specific vulnerabilities, informing both researchers and practitioners about tool selection and expected performance.

7.3 Limitations

Several limitations constrain the scope and generalizability of this research. FlowMauiBlazor's current implementation focuses on a subset of MASVS categories, limiting its complete mobile application security landscape coverage. While effective for targeted analysis, the tool's source and sink definitions rely on pattern-matching approaches that may not capture all possible vulnerability variants.

The evaluation was conducted primarily on IVMAUIB, and broader validation across diverse real-world applications would strengthen the findings' generalizability. The IFDS implementation may not achieve complete soundness across all C# language constructs and .NET library interactions, potentially leading to false negatives in complex scenarios.

Chapter 8

Future Work

The development and evaluation of FlowMauiBlazor have provided insights and a foundational prototype. However, the scope of this thesis naturally leaves several avenues open for future research to create a more comprehensive and robust static analysis solution. The following areas represent promising directions for extending the current work:

8.1 Refinement of IFDS Algorithm Implementation

While the current state of FlowMauiBlazor successfully implements the core IFDS algorithm, specific aspects could be further refined for improved precision and broader applicability:

- **Return-Flow Handling for Multiple Exit Points:** The current model for handling method returns could be extended to more precisely manage methods with multiple return statements or exceptional exit paths. While IFDS traditionally maps to a single abstract exit node, a more thorough approach to propagating facts based on specific return sites could increase analysis precision, especially in complex control flows.

8.2 Scalability and Performance

To ensure the practical applicability of the analyzer to large, real-world .NET MAUI Blazor applications, performance and scalability are key considerations:

- **Performance Optimizations and Parallelization:** FlowMauiBlazor prioritized correctness and precision over speed. Significant performance gains could be achieved by optimizing data structures, refining the worklist algorithm, and exploring more comprehensive parallelization strategies. As demonstrated in other IFDS implementations (e.g., IFDS-A by Rodriguez and Lhoták [55] or IFDS-RA by Ackland [57]), the IFDS algorithm is suitable for concurrent execution, which could drastically reduce analysis time on multi-core processors.
- **Summarization of Library Calls:** A significant challenge for any static analyzer is handling calls to external libraries and the .NET Base Class Library. Analyzing the complete source code of these libraries is often infeasible and can lead to an explosion in the scope of analysis. Future work could focus on developing or integrating techniques for creating precise and sound summaries for common library functions. These summaries would model how tainted facts propagate

through library calls without analyzing their implementations directly, significantly improving scalability and potentially precision if the summaries are accurate. This is a well-established technique in static analysis (e.g., FlowDroid [62], Bodden [56]).

8.3 Taint Model and Vulnerability Coverage

FlowMauiBlazor focuses on a specific set of sources and sinks relevant to Blazor hybrid interoperation. Expanding this model is crucial for broader vulnerability detection:

- **Sophisticated Source, Sink, and Sanitizer Modeling:** The current mechanism for identifying sources and sinks, is based on direct semantic symbol matching. This could be significantly improved by:
  - Implementing a configurable specification language (similar to those used in tools like FlowDroid) allows users or security experts to define custom sources, sinks, and sanitization routines more flexibly.
  - Investigating machine learning techniques to automatically identify potential sources and sinks from code patterns or API usage.
  - Developing more nuanced models for sanitization routines, moving beyond simple taint killing to understand how different sanitizers transform or validate data.

  This would allow the analyzer to be adapted to detect a wider range of vulnerabilities across more MASVS categories, such as SQL injection (by modeling database APIs as sinks and string concatenations as propagation points), insecure network communication, or more subtle cryptographic weaknesses.

8.4 Holistic Hybrid Application Analysis

To provide a truly comprehensive security assessment of .NET MAUI Blazor Hybrid applications, the analysis scope could be broadened:

- **JavaScript Taint Analysis Integration:** FlowMauiBlazor focuses on the C# side of the hybrid application, treating JavaScript interoperability points as sources or sinks. A significant extension would be to integrate a JavaScript taint analyzer. This would enable tracking taint flows that originate in JavaScript, propagate to C#, and potentially flow back to JavaScript. Such bidirectional analysis would provide a more complete picture of vulnerabilities that span the hybrid boundary, which is crucial for frameworks like Blazor where C# and JavaScript are interwoven. This could introduce challenges in analyzing JavaScript and accurately mapping data flows between the two distinct language environments.

Addressing these areas could build upon the foundation laid by this thesis, moving towards a more mature and comprehensive static analysis solution tailored to the unique security challenges of .NET MAUI Blazor Hybrid applications. Such advancements could be valuable for developers seeking to build secure mobile applications using this modern .NET technology.

References

[1] Robert Zeithammer, James Macinko, and Diana Silver, "Assessing the deterrent effects of ignition interlock devices," *American Journal of Preventive Medicine*, vol. 68, no. 1, pp. 137–144, 2025, ISSN: 0749-3797. DOI: https://doi.org/10.1016/j.amepre.2024.09.009.

[2] National Highway Traffic Safety Administration. National Center for Statistics and Analysis, *State alcohol-impaired-driving estimates: 2022 data (traffic safety facts. report no. dot hs 813 579)*, https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/813579, 2024. (visited on 02/11/2025).

[3] Federal Bureau of Investigation, *Crime in the u.s. 2019 – table 29: Estimated number of arrests*, https://ucr.fbi.gov/crime-in-the-u.s/2019/crime-in-the-u.s.-2019/topic-pages/tables/table-29, 2019. (visited on 02/11/2025).

[4] Per Hurtig, "Analys av trafiksäkerhetsutvecklingen 2023 : Målstyrning av trafiksäkerhetsarbetet mot etappmålen 2030," Swedish Transport Administration, Tech. Rep. 2024:091, 2024, p. 47.

[5] Smart Start, *What is an ignition interlock device?* 2016. [Online]. Available: https://www.smartstartinc.com/blog/what-is-an-ignition-interlock-device-iid/ (visited on 02/11/2025).

[6] Willis C, Lybrand S, and Bellamy N, "Alcohol ignition interlock programmes for reducing drink driving recidivism," *Cochrane Database of Systematic Reviews*, no. 3, 2004, ISSN: 1465-1858. DOI: 10.1002/14651858.CD004168.pub2.

[7] Igor Radun, Jussi Ohisalo, Jenni E. Radun Sirpa Rajalin, Mattias Wahde, and Timo Lajunen, "Alcohol ignition interlocks in all new vehicles: A broader perspective," *Traffic Injury Prevention*, vol. 15, no. 4, pp. 335–342, 2014. DOI: 10.1080/15389588.2013.825042.

[8] Bo Lönegren and Liza Jakobsson, "The commercial use of alcohol ignition interlocks," *Alcohol Interlock Programs*, 2005.

[9] Jeroen Willemsen, Carlos Holguera, Bernhard Mueller, Sven Schleier, and Jeroen Beckers, *Mobile application security verification standard (masvs)*, OWASP Foundation, 2024. [Online]. Available: https://mas.owasp.org/MASVS/ (visited on 02/25/2025).

[10] Sungho Lee, Julian Dolby, and Sukyoung Ryu, "Hybridroid: Static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 250–261, ISBN: 9781450338455. DOI: 10.1145/2970276.2970368.

[11] Jali Juhola, "Security testing process for react native applications," *Master's thesis, Tampere University*, 2022.

[12] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein, "Call graph soundness in android static analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and*

Analysis, ser. ISSTA 2024, Vienna, Austria: Association for Computing Machinery, 2024, pp. 945–957, ISBN: 9798400706127. DOI: 10.1145/3650212.3680333. [Online]. Available: https://doi.org/10.1145/3650212.3680333.

[13] Thomas Reps, Susan Horwitz, and Mooly Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61, ISBN: 0897916921. DOI: 10.1145/199448.199462.

[14] The .NET Development Team, *Roslyn (.net compiler platform)*, 2025. [Online]. Available: https://github.com/dotnet/roslyn (visited on 04/28/2024).

[15] TechTarget, *What is firmware?* 2025. [Online]. Available: https://www.techtarget.com/whatis/definition/firmware (visited on 02/12/2025).

[16] Changqing Sun, Renpeng Xing, Yong Wu, Guangxu Zhou, Fuquan Zheng, and Dairong Hu, "Design of over-the-air firmware update and management for iot device with cloud-based restful web services," in *2021 China Automation Congress (CAC)*, 2021, pp. 5081–5085.

[17] Poonam Thakur, Varsha Bodade, Angitha Achary, Madhuri Addagatla, Neeraj Kumar, and Yogesh Pingle, "Universal firmware upgrade over-the-air for iot devices with security," in *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2019, pp. 27–30.

[18] OWASP Foundation, *About OWASP*, 2024. [Online]. Available: https://owasp.org/about/ (visited on 02/27/2025).

[19] OWASP Foundation, *OWASP Mobile App Security Project*, 2024. [Online]. Available: https://owasp.org/www-project-mobile-app-security/ (visited on 02/27/2025).

[20] OWASP, *Mobile application security testing guide (mastg)*, 2025. [Online]. Available: https://mas.owasp.org/MASTG/ (visited on 02/11/2025).

[21] Shivi Garg and Niyati Baliyan, "Comparative analysis of android and ios from security viewpoint," *Computer Science Review*, vol. 40, p. 100 372, 2021, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2021.100372.

[22] SOAX Research Team, *What's android's market share? (updated jan 2025)*, 2025. [Online]. Available: https://soax.com/research/android-market-share (visited on 02/19/2025).

[23] Chiradeep BasuMallick, *Android OS: History, Features, Versions, and Benefits | Spiceworks - Spiceworks — spiceworks.com*, https://www.spiceworks.com/tech/tech-general/articles/android-os/, 2024.

[24] Android Developers, *App Components - Fundamentals*, 2025. [Online]. Available: https://developer.android.com/guide/components/fundamentals (visited on 02/27/2025).

[25] MITRE Corporation, *CWE-926: Improper Export of Android Application Components*, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/926.html (visited on 02/27/2025).

[26] Android Developers, *Usb host and accessory overview*, 2025. [Online]. Available: https://developer.android.com/develop/connectivity/usb (visited on 02/19/2025).

[27] Microsoft, *What is .NET MAUI?* 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-9.0 (visited on 02/27/2025).

[28] Ivano Malavolta, "Beyond native apps: Web technologies to the rescue! (keynote)," ser. Mobile! 2016, Association for Computing Machinery, 2016, pp. 1–2, ISBN: 9781450346436. DOI: 10.1145/3001854.3001863.

[29] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef, and Ayman M. Wahba, "Taxonomy of cross-platform mobile applications development approaches," *Ain Shams Engineering Journal*, vol. 8, no. 2, pp. 163–190, 2017, ISSN: 2090-4479. DOI: https://doi.org/10.1016/j.asej.2015.08.004.

[30] Microsoft, *ASP.NET Core Blazor*, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-9.0 (visited on 02/27/2025).

[31] Stack Overflow, *Stack Overflow Developer Survey 2024 - Technology*, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology (visited on 02/27/2025).

[32] Microsoft, *ASP.NET Core Blazor Hosting Models*, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-9.0 (visited on 02/27/2025).

[33] Microsoft, *Call javascript functions from .net methods in asp.net core blazor*, https://learn.microsoft.com/en-us/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet?view=aspnetcore-9.0, 2025. (visited on 05/15/2025).

[34] Microsoft, *Asp.net core razor component lifecycle*, Mar. 18, 2025. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/lifecycle?view=aspnetcore-9.0 (visited on 02/27/2025).

[35] Microsoft, *Aspńet core blazor event handling*, https://learn.microsoft.com/en-us/aspnet/core/blazor/components/event-handling?view=aspnetcore-9.0, Nov. 2024. (visited on 05/15/2025).

[36] Microsoft, *Asp.net core razor component lifecycle*, Dec. 11, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/data-binding?view=aspnetcore-9.0 (visited on 02/27/2025).

[37] Ashish Aggarwal and Pankaj Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, 2006, pp. 343–350. DOI: 10.1109/COMPSAC.2006.55.

[38] Ryan Dewhurst, *Static Code Analysis*, 2020. [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis (visited on 02/27/2025).

[39] TechMagic, *Dynamic application security testing: The ultimate guide*, 2023. [Online]. Available: https://medium.com/techmagic/dynamic-application-security-testing-the-ultimate-guide-32948e8f138b (visited on 02/27/2025).

[40] Mobile Security Framework (MobSF), *Mobile security framework (mobsf)*, 2025. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF (visited on 02/11/2025).

[41] Narmada Kohli and Mahsa Mohaghegh, "Security testing of android based covid tracer applications," in *2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, 2020, pp. 1–6. DOI: 10.1109/CSDE50874.2020.9411579.

[42] Rudy Chandra *et al.*, "Mobile application security in the health care and finance sector with static analysis (tools: Mobsf)," in *2023 29th International Conference on Telecommunications (ICT)*, 2023, pp. 1–7. DOI: 10.1109/ICT60153.2023.10374057.

[43] Kristóf Tóth, *Allsafe: Intentionally vulnerable android application*, 2025. [Online]. Available: https://github.com/t0thkr1s/allsafe (visited on 03/20/2025).

[44] OWASP ZAP Team, *Getting started with owasp zap*, 2025. [Online]. Available: https://www.zaproxy.org/getting-started/ (visited on 03/20/2025).

[45] Wesley de Kraker, *Xamarin security scanner*, 2024. [Online]. Available: https://github.com/wesleydekraker/xamarin-security-scanner (visited on 04/25/2025).

[46] Security Code Scan Team, *Security code scanner*. [Online]. Available: https://security-code-scan.github.io/ (visited on 04/25/2025).

[47] SonarSource, *Sonarqube*, 2024. [Online]. Available: https://www.sonarsource.com/products/sonarqube/ (visited on 04/25/2025).

[48] SonarQube, *Managing security hotspots*. [Online]. Available: https://docs.sonarsource.com/sonarqube-server/latest/user-guide/security-hotspots/ (visited on 04/25/2025).

[49] InfosecTrain, *Common weakness enumeration: Why is it important?* Oct. 13, 2023. [Online]. Available: https://medium.com/@Infosec-Train/common-weakness-enumeration-why-is-it-important-ef231635966a (visited on 02/27/2025).

[50] Terry Lewis, *What is a CVE Vulnerability and why are they important? — roboshadow.com*, https://www.roboshadow.com/blog/what-are-cve-vulnerabilites, 2023. (visited on 03/20/2025).

[51] MITRE Corporation, *Common vulnerabilities and exposures (cve)*, 2025. [Online]. Available: https://cve.mitre.org/ (visited on 03/20/2025).

[52] King Night, *Understanding the differences between cve, cwe, and nvd*, 2025. [Online]. Available: https://medium.com/@King_Night/understanding-the-differences-between-cve-cwe-and-nvd-e2db34633da8.

[53] MITRE Corporation, *Common weakness enumeration (cwe)*, 2025. [Online]. Available: https://cwe.mitre.org/about/index.html (visited on 03/20/2025).

[54] Puzhuo Liu *et al.*, "Llm-powered static binary taint analysis," vol. 34, no. 3, 2025, ISSN: 1049-331X.

[55] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez, "Practical extensions to the ifds algorithm," in *Compiler Construction*, Rajiv Gupta, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144, ISBN: 978-3-642-11970-5.

[56] Eric Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12, Beijing, China: Association for Computing Machinery, 2012, pp. 3–8, ISBN: 9781450314909. DOI: 10.1145/2259051.2259052.

[57] Patrik Ackland, *Fast and scalable static analysis using deterministic concurrency*, 2017.

[58] The Soot and Heros Development Teams, *Heros: Context-sensitive data-flow analysis framework*. [Online]. Available: https://github.com/soot-oss/heros (visited on 04/28/2025).

[59] The WALA Development Team, *Wala (worklist analysis framework for java)*, 2025. [Online]. Available: https://github.com/wala/WALA (visited on 04/28/2025).

[60] A.D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation," Jan. 2016.

[61] Yonghui Liu *et al.*, "Demystifying react native android apps for static analysis," *ACM Trans. Softw. Eng. Methodol.*, Nov. 2024, Just Accepted, ISSN: 1049-331X. DOI: 10.1145/3702977.

[62] Li Li, Tegawendé F. Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon, "The multi-generation repackaging hypothesis," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17,

Buenos Aires, Argentina: IEEE Press, 2017, pp. 344–346. DOI: 10.1109/ICSE-C.2017.140.

[63] OWASP Foundation, *Owasp benchmark project*, https://owasp.org/www-project-benchmark/, 2022. (visited on 04/28/2024).

[64] Fabian Yamaguchi, Nils Golde, Daniel Arp, and Konrad Rieck, "Modeling and discovering vulnerabilities with code property graphs," *IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

[65] Dejan Baca, Martin Boldt, and Bengt Carlsson, "A novel method for evaluating the quality of security static analysis tools," in *International Conference on Availability, Reliability and Security (ARES)*, 2017, pp. 1–10.

[66] Robby Johnson, Juan Caballero, and Somesh Jha, "Analysis of vulnerability scanners and static analysis tools: A survey," in *IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 1–10.

Appendix A

Prototype IFDS Taint Analyzer Report

The following is the taint analysis report output generated by the prototype IFDS taint analyzer, FlowMauiBlazor, when run against the vulnerable application IVMAUIB. It reports the instances of tainted value data flowing into a defined sink.

```
============================================================
              TAINT ANALYSIS REPORT
============================================================

[Analysis Statistics]
--------------------
- Analysis Duration: 00:00:08.6117508
- Vulnerabilities Found: 14

[POTENTIAL VULNERABILITIES DETECTED]
---------------------------

--- Finding #1 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_CryptoVulnerabilities_razor.g.cs(607,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: Components_Pages_CryptoVulnerabilities_razor.g.cs:L607 (In
        ↪ Method: Task CryptoVulnerabilities.StoreWithHardcodedKey(), Kind:
        ↪ CallSite)
    Sink Operation: Console.WriteLine($"Stored encrypted data using hardcoded
        ↪ key: {encrypted}");
    Original Taint: ZeroFact { }
      at Location: Components_Pages_CryptoVulnerabilities_razor.g.cs:L601

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_CryptoVulnerabilities_razor.g.cs:L601] ->
          ↪ encrypted = CryptoService.Encrypt(sensitiveData) (ZeroFact { })
      [THROUGH][InsecureCryptoService.cs:L18 (In Method: string
          ↪ InsecureCryptoService.Encrypt(string plainText), Kind: Entry)] ->
          ↪ Entry to method string InsecureCryptoService.Encrypt(string
          ↪ plainText) (AccessPath { Base = string plainText, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })
      [THROUGH][InsecureCryptoService.cs:L18 (In Method: string
          ↪ InsecureCryptoService.Encrypt(string plainText), Kind: Exit)] ->
          ↪ Exit from method string InsecureCryptoService.Encrypt(string
          ↪ plainText) (AccessPath { Base = string plainText, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })
```

```
      [THROUGH][Components_Pages_CryptoVulnerabilities_razor.g.cs:L602] ->
          ↪ DataStorageService.SaveInsecurely("hardcoded_key_data", encrypted)
          ↪ ; (AccessPath { Base = encrypted, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_CryptoVulnerabilities_razor.g.cs:L604] ->
          ↪ resultMessage = "Data encrypted with hardcoded key and stored"; (
          ↪ AccessPath { Base = encrypted, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_CryptoVulnerabilities_razor.g.cs:L607] ->
          ↪ Console.WriteLine($"Stored encrypted data using hardcoded key: {
          ↪ encrypted}"); (AccessPath { Base = encrypted, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })

----------------------

--- Finding #2 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_CryptoVulnerabilities_razor.g.cs(655,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: Components_Pages_CryptoVulnerabilities_razor.g.cs:L655 (In
        ↪ Method: Task CryptoVulnerabilities.EncryptWithPassword(), Kind:
        ↪ CallSite)
    Sink Operation: Console.WriteLine($"Used password '{password}' to derive
        ↪ key '{weakKey}'");
    Original Taint: ZeroFact { }
      at Location: Components_Pages_CryptoVulnerabilities_razor.g.cs:L645

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_CryptoVulnerabilities_razor.g.cs:L645] ->
          ↪ weakKey = CryptoService.Hash(password) (ZeroFact { })
      [THROUGH][InsecureCryptoService.cs:L28 (In Method: string
          ↪ InsecureCryptoService.Hash(string input), Kind: Entry)] -> Entry
          ↪ to method string InsecureCryptoService.Hash(string input) (
          ↪ AccessPath { Base = string input, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][InsecureCryptoService.cs:L28 (In Method: string
          ↪ InsecureCryptoService.Hash(string input), Kind: Exit)] -> Exit
          ↪ from method string InsecureCryptoService.Hash(string input) (
          ↪ AccessPath { Base = string input, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_CryptoVulnerabilities_razor.g.cs:L648] ->
          ↪ DataStorageService.SaveInsecurely("password_protected_data", $"{
          ↪ weakKey}:{CryptoService.Encrypt(p... (AccessPath { Base = weakKey,
          ↪  Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_CryptoVulnerabilities_razor.g.cs:L651] ->
          ↪ resultMessage = "Data encrypted with password-derived key (weak
          ↪ derivation)"; (AccessPath { Base = weakKey, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })
      [THROUGH][Components_Pages_CryptoVulnerabilities_razor.g.cs:L652] ->
          ↪ passwordResult = "Encrypted and stored"; (AccessPath { Base =
          ↪ weakKey, Fields = System.Collections.Immutable.ImmutableArray`1[
          ↪ Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_CryptoVulnerabilities_razor.g.cs:L655] ->
          ↪ Console.WriteLine($"Used password '{password}' to derive key '{
          ↪ weakKey}'"); (AccessPath { Base = weakKey, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })

----------------------

--- Finding #3 ---
```

```
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(350,17)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'args' (index 2) of sink
        ↪ method 'ValueTask<string> JSRuntimeExtensions.InvokeAsync<string>(
        ↪ IJSRuntime jsRuntime, string identifier, params object?[]? args)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L350 (In
        ↪ Method: Task InteropVulnerabilities.ExecuteJavaScript(), Kind:
        ↪ CallSite)
    Sink Operation: result = await JSRuntime.InvokeAsync<string>("eval",
        ↪ jsCodeToExecute)
    Original Taint: ZeroFact { }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L349

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L349] ->
          ↪ jsCodeToExecute = jsCode (ZeroFact { })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L350] ->
          ↪ result = await JSRuntime.InvokeAsync<string>("eval",
          ↪ jsCodeToExecute) (AccessPath { Base = jsCodeToExecute, Fields =
          ↪ System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #4 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(381,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L381 (In
        ↪ Method: string InteropVulnerabilities.ExecuteCommandFromJs(string
        ↪ command), Kind: CallSite)
    Sink Operation: Console.WriteLine($"Executed command from JavaScript: {
        ↪ command}");
    Original Taint: AccessPath { Base = string command, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L375 (In
          ↪ Method: string InteropVulnerabilities.ExecuteCommandFromJs(string
          ↪ command), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L375 (In
          ↪ Method: string InteropVulnerabilities.ExecuteCommandFromJs(string
          ↪ command), Kind: Entry)] -> Entry to method string
          ↪ InteropVulnerabilities.ExecuteCommandFromJs(string command) (
          ↪ AccessPath { Base = string command, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L380] ->
          ↪ result = NativeFeatures.ExecuteCommand(command).Result (AccessPath
          ↪  { Base = string command, Fields = System.Collections.Immutable.
          ↪ ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L381] ->
          ↪ Console.WriteLine($"Executed command from JavaScript: {command}");
          ↪  (AccessPath { Base = string command, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #5 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(397,13)
```

```
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'key' (index 0) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L397 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Preferences.Default.Set($"js_{type}_username", username);
    Original Taint: AccessPath { Base = string type, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string type,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string type, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #6 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(397,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 1) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L397 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Preferences.Default.Set($"js_{type}_username", username);
    Original Taint: AccessPath { Base = string username, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string username,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string username, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #7 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(398,13)
```

```
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'key' (index 0) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L398 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Preferences.Default.Set($"js_{type}_password", password);
    Original Taint: AccessPath { Base = string type, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string type,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string type, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L398] ->
          ↪ Preferences.Default.Set($"js_{type}_password", password); (
          ↪ AccessPath { Base = string type, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #8 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(398,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 1) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L398 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Preferences.Default.Set($"js_{type}_password", password);
    Original Taint: AccessPath { Base = string password, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string password,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string password, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L398] ->
          ↪ Preferences.Default.Set($"js_{type}_password", password); (
          ↪ AccessPath { Base = string password, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
```

```
----------------------

--- Finding #9 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(399,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L399 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Console.WriteLine($"Stored {type} credentials from
        ↪ JavaScript: {username}");
    Original Taint: AccessPath { Base = string username, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string username,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string username, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L398] ->
          ↪ Preferences.Default.Set($"js_{type}_password", password); (
          ↪ AccessPath { Base = string username, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L399] ->
          ↪ Console.WriteLine($"Stored {type} credentials from JavaScript: {
          ↪ username}"); (AccessPath { Base = string username, Fields = System
          ↪ .Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })

----------------------

--- Finding #10 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\obj\Debug\net9.0-android\Microsoft.CodeAnalysis.Razor
    ↪ .Compiler\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator\
    ↪ Components_Pages_InteropVulnerabilities_razor.g.cs(399,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L399 (In
        ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
        ↪ username, string password, string type), Kind: CallSite)
    Sink Operation: Console.WriteLine($"Stored {type} credentials from
        ↪ JavaScript: {username}");
    Original Taint: AccessPath { Base = string type, Fields = System.
        ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
        ↪ IFieldSymbol] }
      at Location: Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)

  Taint Propagation Trace:
```

```
      [SOURCE] [Components_Pages_InteropVulnerabilities_razor.g.cs:L392 (In
          ↪ Method: void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string password, string type), Kind: Entry)] -> Entry to
          ↪  method void InteropVulnerabilities.StoreCredentialsFromJs(string
          ↪ username, string passwo... (AccessPath { Base = string type,
          ↪ Fields = System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L397] ->
          ↪ Preferences.Default.Set($"js_{type}_username", username); (
          ↪ AccessPath { Base = string type, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][Components_Pages_InteropVulnerabilities_razor.g.cs:L398] ->
          ↪ Preferences.Default.Set($"js_{type}_password", password); (
          ↪ AccessPath { Base = string type, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [Components_Pages_InteropVulnerabilities_razor.g.cs:L399] ->
          ↪ Console.WriteLine($"Stored {type} credentials from JavaScript: {
          ↪ username}"); (AccessPath { Base = string type, Fields = System.
          ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
          ↪ IFieldSymbol] })

----------------------

--- Finding #11 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\Services\InsecureAuthService.cs(44,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 1) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: InsecureAuthService.cs:L44 (In Method: void
        ↪ InsecureAuthService.SaveAuthToken(string token), Kind: CallSite)
    Sink Operation: Preferences.Default.Set("auth_token", token);
    Original Taint: ZeroFact { }
      at Location: Components_Pages_AuthVulnerabilities_razor.g.cs:L569

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_AuthVulnerabilities_razor.g.cs:L569] ->
          ↪ AuthService.SaveAuthToken(authToken); (ZeroFact { })
      [THROUGH][InsecureAuthService.cs:L41 (In Method: void InsecureAuthService
          ↪ .SaveAuthToken(string token), Kind: Entry)] -> Entry to method
          ↪ void InsecureAuthService.SaveAuthToken(string token) (AccessPath {
          ↪  Base = string token, Fields = System.Collections.Immutable.
          ↪ ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [InsecureAuthService.cs:L44] -> Preferences.Default.Set("
          ↪ auth_token", token); (AccessPath { Base = string token, Fields =
          ↪ System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #12 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\Services\InsecureAuthService.cs(48,13)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'contents' (index 1) of sink
        ↪ method 'void File.WriteAllText(string path, string? contents)'.
    Sink Location: InsecureAuthService.cs:L48 (In Method: void
        ↪ InsecureAuthService.SaveAuthToken(string token), Kind: CallSite)
    Sink Operation: File.WriteAllText(path, token);
    Original Taint: ZeroFact { }
      at Location: Components_Pages_AuthVulnerabilities_razor.g.cs:L569

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_AuthVulnerabilities_razor.g.cs:L569] ->
          ↪ AuthService.SaveAuthToken(authToken); (ZeroFact { })
```

```
      [THROUGH][InsecureAuthService.cs:L41 (In Method: void InsecureAuthService
          ↪ .SaveAuthToken(string token), Kind: Entry)] -> Entry to method
          ↪ void InsecureAuthService.SaveAuthToken(string token) (AccessPath {
          ↪  Base = string token, Fields = System.Collections.Immutable.
          ↪ ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [THROUGH][InsecureAuthService.cs:L44] -> Preferences.Default.Set("
          ↪ auth_token", token); (AccessPath { Base = string token, Fields =
          ↪ System.Collections.Immutable.ImmutableArray`1[Microsoft.
          ↪ CodeAnalysis.IFieldSymbol] })
      [THROUGH][InsecureAuthService.cs:L47] -> path = Path.Combine(Environment.
          ↪ GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "t
          ↪ ... (AccessPath { Base = string token, Fields = System.Collections
          ↪ .Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol]
          ↪ })
      [SINK]   [InsecureAuthService.cs:L48] -> File.WriteAllText(path, token);
          ↪ (AccessPath { Base = string token, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #13 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\Services\InsecureDataStorageService.cs(82,17)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 0) of sink
        ↪ method 'void Console.WriteLine(string? value)'.
    Sink Location: InsecureDataStorageService.cs:L82 (In Method: void
        ↪ InsecureDataStorageService.SaveInsecurely(string key, string value),
        ↪  Kind: CallSite)
    Sink Operation: Console.WriteLine($"Saving sensitive data: {key}={value}");
    Original Taint: ZeroFact { }
      at Location: Components_Pages_StorageVulnerabilities_razor.g.cs:L324

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_StorageVulnerabilities_razor.g.cs:L324] ->
          ↪ DataStorageService.SaveInsecurely("username", username); (ZeroFact
          ↪  { })
      [THROUGH][InsecureDataStorageService.cs:L75 (In Method: void
          ↪ InsecureDataStorageService.SaveInsecurely(string key, string value
          ↪ ), Kind: Entry)] -> Entry to method void
          ↪ InsecureDataStorageService.SaveInsecurely(string key, string value
          ↪ ) (AccessPath { Base = string value, Fields = System.Collections.
          ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
      [SINK]   [InsecureDataStorageService.cs:L82] -> Console.WriteLine($"
          ↪ Saving sensitive data: {key}={value}"); (AccessPath { Base =
          ↪ string value, Fields = System.Collections.Immutable.ImmutableArray
          ↪ `1[Microsoft.CodeAnalysis.IFieldSymbol] })

----------------------

--- Finding #14 ---
WARNING [TAINT0001] at C:\Users\miles\source\repos\InsecureMauiBlazor\
    ↪ InsecureMauiBlazor\Services\InsecureDataStorageService.cs(85,17)
  Title: Tainted Value Reaches Sink
  Taint vulnerability detected!
    Description: Tainted data flows into argument 'value' (index 1) of sink
        ↪ method 'void IPreferences.Set<string>(string key, string value,
        ↪ string? sharedName = null)'.
    Sink Location: InsecureDataStorageService.cs:L85 (In Method: void
        ↪ InsecureDataStorageService.SaveInsecurely(string key, string value),
        ↪  Kind: CallSite)
    Sink Operation: Preferences.Default.Set(key, value);
    Original Taint: ZeroFact { }
      at Location: Components_Pages_StorageVulnerabilities_razor.g.cs:L324

  Taint Propagation Trace:
      [SOURCE] [Components_Pages_StorageVulnerabilities_razor.g.cs:L324] ->
          ↪ DataStorageService.SaveInsecurely("username", username); (ZeroFact
          ↪  { })
```

```
[THROUGH][InsecureDataStorageService.cs:L75 (In Method: void
    ↪ InsecureDataStorageService.SaveInsecurely(string key, string value
    ↪ ), Kind: Entry)] -> Entry to method void
    ↪ InsecureDataStorageService.SaveInsecurely(string key, string value
    ↪ ) (AccessPath { Base = string value, Fields = System.Collections.
    ↪ Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.IFieldSymbol] })
[THROUGH][InsecureDataStorageService.cs:L82] -> Console.WriteLine($"
    ↪ Saving sensitive data: {key}={value}"); (AccessPath { Base =
    ↪ string value, Fields = System.Collections.Immutable.ImmutableArray
    ↪ `1[Microsoft.CodeAnalysis.IFieldSymbol] })
[SINK]   [InsecureDataStorageService.cs:L85] -> Preferences.Default.Set(
    ↪ key, value); (AccessPath { Base = string value, Fields = System.
    ↪ Collections.Immutable.ImmutableArray`1[Microsoft.CodeAnalysis.
    ↪ IFieldSymbol] })
```

LISTING A.1: Full Taint Analysis Report from Prototype IFDS Analyzer