VULNERABILITY ASSESSMENT AND POLICY ENFORCEMENT FOR HYBRID MOBILE APPLICATIONS

by

Abhinav Mohanty

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2021

Approved by:

Dr. Meera Sridhar, Chair

Dr. Tom Moyer

Dr. Harini Ramaprasad

Dr. Erik Saule

Dr. Weichao Wang

ABSTRACT

ABHINAV MOHANTY. Vulnerability Assessment and Policy Enforcement for Hybrid Mobile Applications. (Under the direction of DR. MEERA SRIDHAR, CHAIR)

Hybrid mobile apps are becoming increasingly popular for building cross-platform mobile applications, where the core business code of apps is written using web technologies, such as HTML, JavaScript (JS), and Cascading Style Sheets (CSS). This technology allows mobile apps to be *write-once-run-anywhere*, saving substantial time and resources required to develop different apps for different mobile platforms, such as Android and iOS. Hybrid mobile apps are also a lucrative solution for IoT vendors, to assist them in the time-constrained race for market share and provide a quick solution to design cross-platform companion IoT mobile apps to accompany the IoT devices.

However, the fusion of web technologies with the mobile platform also exposes mobile apps to web attacks. Moreover, the inclusion of JavaScript, a powerful and complex scripting language, is dangerous since there is no mechanism to determine the origin (party) of the code to control access. Existing solutions are either limited to a particular platform (e.g., Android) or a specific hybrid framework (e.g., Cordova) or only protect the device resources and disregard the sensitive elements in the web environment. Furthermore, most solutions require modification of the base platform.

The main objective of this dissertation is to provide a comprehensive security solution for hybrid mobile apps. This is achieved through three thrusts—(i) building a flexible, fine-grained, principal-based policy enforcement framework for hybrid mobile apps, capable of protecting against a large class of attacks, retroactively, and without modifying underlying operating systems or development frameworks; (ii) building an automated security assessment framework for hybrid smart home companion apps that can be used by developers or third-parties to assess hybrid apps for preexisting security issues; and (iii) finally, building a web-based framework that can be used to teach advance cybersecurity skills including

concepts of hybrid app security.

DEDICATION

I would dedicate my dissertation to my parents Premananda Mohanty and Dr. Manju Mohanty for a lifetime of support and guidance. I would also dedicate my dissertation to my grandmother Jambu Lata Mallick and my uncle Vijay Kumar Mallick. Their constant support made me the person I am today. I would also like to acknowledge my fiancé Mahak Malik for being extremely supportive and patient throughout my doctoral studies. Without these people, my dissertation journey would be unbearable.

I would like to appreciate my friends here in the United States who eventually became family—Nishant Gaurav, Geetanjali Pathak, Nikhil Sanil, Rakshit Rathi, Sangram Sabnis, Mohit Arora, Alankar Padman, Saurabh Landge, Karishma Borole, Gatha Sehgal, Anusha Iyer, Sunakshi Sharma, Rajendra Meena, and Mary Ann Baldo-Meena, Shantanu Rajenimbalkar, Hemanth Satish Kumar, Vishwanath Rana, and Bhavesh Tadvi, for constantly being there through all the ups and downs in this journey.

And finally, I would like to acknowledge my childhood friends Abhishek Sood, Arya Mitra Malik, Piyush Chandra, Arjun Nanda, Ankit Walia, Prateek Bathla, Aman Mukhija, Ankit Sharma, and Pavinayan Sharma for being a constant in my life.

ACKNOWLEDGEMENTS

Throughout the writing of this dissertation I have received a great deal of support and assistance.

I would first like to thank my advisor, Dr. Meera Sridhar, whose expertise was invaluable in formulating the research questions and methodology. Her insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Her unwavering guidance, relentless support and patience throughout my doctoral studies cannot be underestimated. Under her supervision, I improved not only as a researcher but also in technical writing, collaborating, verbally explaining my ideas, and overall as a person. She is the greatest teacher I will ever get.

I would also like to express my sincere gratitude to my dissertation committee members: Dr. Tom Moyer, Dr. Harini Ramaprasad, and Dr. Erik Saule, and Dr. Weichao Wang for serving on my dissertation committee and providing guidance and suggestions to improve my work. Your suggestions are extremely valuable.

My journey would have been unbearable without the constant support and guidance of my colleagues Fadi Yilmaz, Islam Obaidat, and Amirreza Niakanlahiji. The brotherly bond that I established with them will last forever.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CSP Content Security Policy

CSRF Cross-site Request Forgery

CSS Cascading Style Sheets

HTML Hyper Text Markup Language

IoT Internet of Things

IRM In-lined reference monitor/monitoring

JS JavaScript

OS Operating System

SOP Same-origin Policy

SQL Structured Query Language

SQLi SQL Injection

XSS Cross-site Scripting

CHAPTER 1: INTRODUCTION

Hybrid mobile apps are *cross-platform* mobile applications developed using web technologies. The app developer uses HTML and JavaScript to write the core business code of the app and uses CSS (Cascading Style Sheets) to design and style the app [2]. *Hybrid app development frameworks* automatically package the core code within a native app container for a particular mobile platform, such as Android and iOS. This technology allows mobile apps to be *write-once-run-anywhere*, which saves substantial time and resources to develop different versions of the same app for different mobile platforms. Hybrid apps execute inside an app embedded web browser since they use web technologies.

The advent of near desktop-quality processors in smartphones and smartphone RAM ranging from 6 GB to 12 GB [3], mobile OSes becoming more robust than earlier, and improvements in the performance of JavaScript engines have significantly reduced the performance gap between native and hybrid mobile apps. In past years, hybrid mobile apps suffered numerous issues, such as substandard performance compared to native apps, lack of UI design features supported by native app development platforms, a limited set of tools for app development, and poor user-experience [4, 5]. Apache Cordova and PhoneGap were the most popular frameworks for building hybrid mobile apps. However, in the past few years, apart from the continual development of Cordova and PhoneGap, several new hybrid app development frameworks have emerged. A few popular and recent hybrid app development frameworks include React Native [6], Ionic [7], Framework7 [8], Flutter [9], Onsen UI [10], NativeScript [11], Xamarin [12], and Mobile Angular UI [13]. Ionic's 2020 survey of over 1,700 enterprise developers, architects, and IT leaders indicates that the hybrid approach to developing mobile apps is rapidly gaining ground over native apps [14]. According to the survey, only 7% of surveyed developers exclusively developed native apps in 2020. The

survey also highlights that in an enterprise setting, web developers are developing majority of mobile apps using cross-platform tools (74% correspondents), rather than dedicated mobile developers (17% correspondents) [14].

Like native mobile apps, hybrid apps require access to native OS APIs to access device resources, such as geolocation, contacts, SMS, Bluetooth, camera, NFC, Wireless, File System, device motion and orientation, and Gallery. In the case of Android, these are Java APIs and in the case of iOS, Objective-C/Swift APIs. Hybrid app development frameworks provide *plugins* (also known as *bridges*)—a combination of *web APIs* and *native APIs*, to facilitate this requirement. Using these plugins, the developer can access native functionality, with the low-level OS-specific details abstracted away.

Unfortunately, web content, especially JavaScript, a powerful scripting language, and the complexity of the hybrid software stack tend to introduce subtle security holes that drastically increase the attack surface and exacerbate the security issues in hybrid apps. Plugins allow malicious web entities to access device-level resources; the advent of IoT further increases the attack surface by allowing web content in IoT companion mobile apps to access and control consumer-facing IoT devices. Numerous previous works demonstrate that the inclusion of web content renders hybrid apps vulnerable to various web attacks, such as *code-injection*, *XSS*, *SQL injection*, *fracking*, *data-exfiltration*, and *malvertisements*.

Existing mobile OS permission models (iOS and Android) are too coarse-grained and only implement an *allow/disallow* permission model to prevent the misuse of the plugins. However, once a user grants an app permission to access a device resource, the OS cannot track or control *how* the web content uses the device resource. As an example, let us consider a weather app that is hybrid and free to use. The app requires access to geolocation (device resource) to display the weather at the user's current location. Let us assume the app is overprivileged (requests more permissions than required) and also requests access to the SMS device resource; over-privilege is a common scenario in mobile apps [15, 16, 17]. Once the user grants the requested permissions to the app, the app can track the user's location in

real-time and use the SMS channel to stealthily exfiltrate this sensitive information and send it to an attacker-controlled server.

Existing JavaScript security solutions [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31] cannot be easily adapted to reduce this attack surface because even though the plugins have a significant JavaScript component, they also have a Java component acting as a bridge between the JavaScript code and underlying OS APIs, making them different from standard JavaScript web APIs. Numerous device-related channels, such as SMS, Email or Contacts, cannot be protected by these solutions without significant modifications to the solution. Previous solutions that protect hybrid mobile apps are either limited to a particular platform (e.g., Android) [32, 17, 33, 34] or a specific hybrid framework (e.g. Cordova) [16, 35, 36, 37, 38] or only protect the device resources and disregard the sensitive elements in the web environment [39], such as web APIs of the *Document Object Model* (DOM) [40]. Moreover, most of the solutions require the modification of the base platform [39, 32, 17, 33, 34]. Ideally, a defense solution should provide retroactive protection, be independent of the underlying OS or the hybrid app development framework, and protect access to both—device resources and sensitive DOM objects. None of the existing solutions incorporate all these capabilities.

Additionally, there is exponential growth in the production of *Internet-of-Things* (IoT) devices, such as smart–wearables, TVs, assistant speakers, household appliances, and other consumers electronic devices [41, 42]. The rapid increase in the number of such IoT devices parallels with a certain time-constrained race among the IoT manufacturers for market share. Since *companion* accompany most of these IoT devices, hybrid apps provide a lucrative opportunity to IoT manufacturers to get ahead in this time-constrained race by saving the substantial time required to develop different apps for different platforms. Currently, only a few works focus on securing IoT companion apps [43, 44, 45, 46, 47, 48], and none of the existing works do a security assessment of hybrid companion apps or provide any security solution for these apps.

Finally, as IoT devices and their companion apps grow exponentially in number, and

significantly add to the cyber attack surface, cybersecurity education in mobile app security and IoT security in becoming critical for increasing awareness and improving the workforce. Currently, several gaps require filling in advanced cybersecurity education [49]. For instance, addressing the severe lack of gender and ethnic diversity in the cybersecurity industry is desperately required to meet the growing demand and foster innovation and creativity in problem solving [50, 51, 52, 53]. In order to address these issues, it is important to deliver cybersecurity educational content in an engaging, inclusive way [54].

The main objective of our dissertation is to provide a comprehensive security framework for hybrid mobile apps. We achieve this by building a flexible, fine-grained, principal-based policy enforcement framework for hybrid mobile apps, capable of protecting against a large class of attacks, retroactively, and without modifying underlying operating systems or development frameworks. We also build an automated security assessment framework for hybrid apps that can be used by developers or third-parties to assess hybrid apps for preexisting security issues. Finally, we build a web-based framework that can be used to teach advance cybersecurity skills including concepts of hybrid mobile app security in an interactive, engaging, and inclusive way.

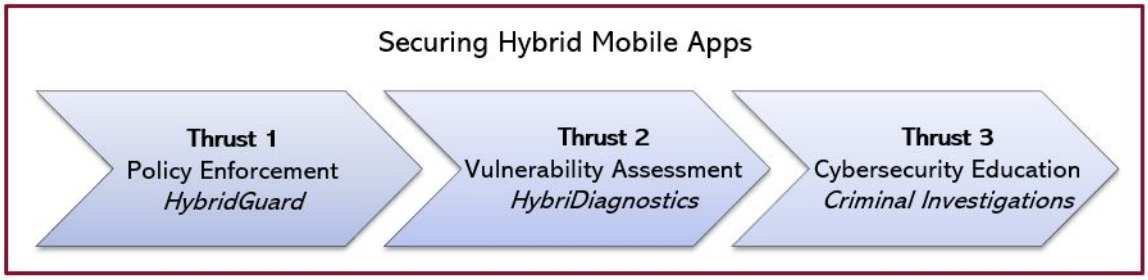

Figure 1.1: Three thrusts of our dissertation

Figure 1.1 demonstrates the three thrusts of our dissertation:

1. In the first thrust (Chapter 3 and 4), we design HybridGuard [55, 56], a robust security enforcement framework for hybrid mobile apps that allows developers (at the app development stage) or any third party (retroactively) to enforce a wide-range of principal

based fine-grained security policies to mitigate attacks originating from JavaScript included by the developer. We ensure that our solution provides retroactive protection, is independent of the underlying OS or the hybrid app development framework, and protects access to both—device resources and sensitive DOM objects. We also design a simple policy enforcing language that allows developers and third-parties to specify principal-based fine-grained policies to be enforced on hybrid mobile apps. We test HybridGuard's compatibility with other hybrid app development frameworks and present the results. We evaluate HybridGuard on real-world apps, and also conduct performance and overhead evaluation. Finally, we also provide the end-user with the capability to customize any policy.

2. In the second thrust (Chapter 5), we design HybriDiagnostics [57], an automated vulnerability-assessment framework that identifies eleven preexisting security issues in hybrid mobile apps. At the heart of HybriDiagnostics is an analysis engine that identifies misconfigured policies (including Content Security Policy, and whitelist), usage of inline scripts, unsafe `eval()` usage, unsafe HTML and JQuery APIs and attributes, unencrypted storage, usage of vulnerable Cordova SDKs, and others. The results of the analyses are documented in a security assessment report.

3. In the third thrust (Chapter 6), we design Criminal Investigations, a gamified, scalable web-based framework for teaching and assessing cybersecurity skills. We envision Criminal Investigations packaged as a series of stackable cybersecurity activities covering topics from the field of hybrid mobile app security and IoT firmware security. Criminal Investigations promotes student engagement and learning by incorporating gamification concepts such as storytelling, experience points, just-in-time learning content delivery and checkpoints into activity design.

1.1 Research Questions

In this section, we present some of the investigative research questions that motivated this dissertation work. Each research question is accompanied by a chapter number, which is the chapter where the respective research question is answered in detail. We also summarize the answers to these questions in §7 (CONCLUSIONS).

1. What types of cyberattacks can originate from the inclusion of third-party JavaScript in hybrid mobile apps? **[Chapter 3]**

2. To what extent do security mechanisms built into the mobile OS, or provided by the embedded browser, or provided by the hybrid app frameworks provide security for hybrid apps from cyber attacks originating from the inclusion of third-party content in hybrid apps? **[Chapter 2]**

3. What are the most prevalent security issues in hybrid mobile apps? **[Chapter 5]**

4. Can in-lined reference monitoring provide an elegant solution for protecting against attacks on user privacy in hybrid mobile apps (especially privacy attacks originating from third-party JavaScript)? **[Chapter 3 and 4]**

5. What are the challenges of designing a secure IRM framework in the cross-domain platform (HTML, CSS, JavaScript) of hybrid mobile apps? **[Chapter 3]**

6. What are the classes of security policies that can be enforced by such an IRM framework? **[Chapter 3 and 4]**

7. How should the policy specification language or platform be designed to also allow users to define the policy? **[Chapter 4]**

8. What is the impact on performance of an app after integrating the IRM framework and enforcing policies? **[Chapter 4]**

9. Which hybrid app development frameworks should we target the IRM framework to be compatible with? **[Chapter 4]**

10. How can security issues in smarthome companion hybrid mobile apps be exploited to attack a smart home ecosystem? **[Chapter 5]**

11. Does gamification help in improving student engagement and learning in advanced cybersecurity topics? **[Chapter 6]**

    - Does using a narrative increases the student's interest in the activity and capture their attention?
    - Does earning experience points (XP) for solving activity challenges motivate the student to perform well in the activity?
    - Does the design of the activity, i.e., colors, fonts, and placement of UI elements follow accessibility principles?

1.2 Publications

Parts of this dissertation have published in various venues. This list includes:

1. HybriDiagnostics: Evaluating Security Issues in Hybrid SmartHome Companion Apps. Abhinav Mohanty, Meera Sridhar. In *IEEE Workshop on the Internet of Safe Things*. April, 2021. Submitted to the Journal *Computers & Security*.

2. HybriDiagnostics: Evaluating Security Issues in Hybrid SmartHome Companion Apps. Abhinav Mohanty, Meera Sridhar. In *IEEE Workshop on the Internet of Safe Things*. May, 2021.

3. A multi-party, fine-grained permission and policy enforcement framework for hybrid mobile applications. Phu H Phung, Rakesh SV Reddy, Steven Cap, Anthony Pierce, Abhinav Mohanty, Meera Sridhar. In the *Journal of Computer Security*, Volume 28, Issue 3, 375–404. April, 2020.

4. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. Phu H. Phung, Abhinav Mohanty, Rahul Rachapalli, Meera Sridhar. In Proceedings of *Mobile Security Technologies (MOST)*, 147–156. May, 2017.

5. POSTER: Criminal Investigations: An Interactive Experience to Improve Student Engagement and Achievement in Cybersecurity courses. Abhinav Mohanty, Pooja Murarisetty, Ngoc Diep Nguyen, Julio César Bahamon, Harini Ramaprasad, Meera Sridhar. Poster presented in the *52nd ACM Technical Symposium on Computer Science Education (SIGCSE)*, 1276–1276. March, 2021.

6. POSTER: Class-sourced Penetration Testing of IoT Devices. Abhinav Mohanty, Parag Mhatre, Meera Sridhar. Poster presented in the *IEEE Workshop on the Internet of Safe Things*. May, 2020.

7. POSTER: Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. Phu H. Phung, Abhinav Mohanty, Rahul Rachapalli, Meera Sridhar. Poster presented in *The Network and Distributed System Security Symposium (NDSS)*. February, 2018.

1.3 Roadmap

The rest of the dissertation is organized as follows. Chapter 2 presents background and discusses the related work. Chapter 3 and Chapter 4 present HybridGuard, our security enforcement framework, discusses the different types of policies that the developer can enforce, framework's implementation, and evaluation. Chapter 5 presents HybriDiagnostics, our automated vulnerability assessment framework that can identify preexisting security issues in hybrid companion apps. Chapter 6 presents Criminal Investigations, our gamified, interactive, and scalable web-based framework for teaching cybersecurity skills, and Chapter 7 concludes by summarizing the answers to the research questions mentioned in §1.1.

CHAPTER 2: BACKGROUND & RELATED WORK

In this section, we discuss the differences between native and hybrid apps, the Apache Cordova hybrid app development framework and other frameworks that extend Cordova, basic security options available to the hybrid app developer, and present a brief background on the typical smarthome ecosystem where companion apps play a crucial role.

2.1 Native vs. Hybrid Mobile Apps

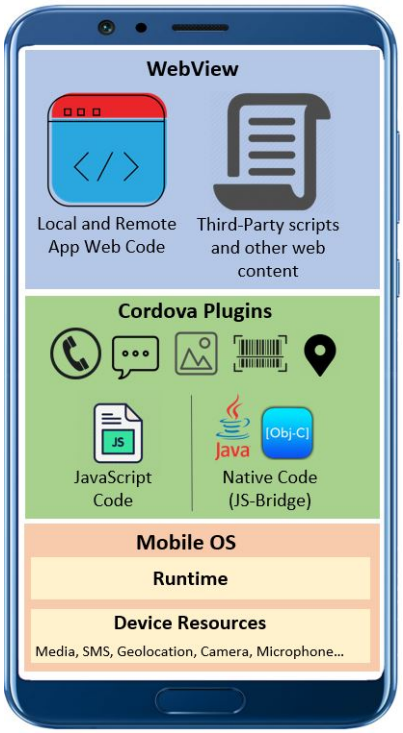

Figure 2.1: Architecture of hybrid mobile apps.

Native apps target a specific operating system or platform and use platform-specific programming language for development, e.g., Java or Kotlin for Android, and Objective-C or Swift for iOS [58]. Native apps use the underlying platform's SDKs to directly access device resources such as geolocation, contacts, camera, microphone, media, SMS, and call

functionality. Native apps also render the app UI using *native* UI components, i.e., interfaces, classes, and methods that belong to the platform's SDK, instead of using web technologies. This tight coupling with the underlying OS makes native apps perform better than hybrid apps in resource access speed and UI transitions [59]. However, native apps have high development and maintenance costs since they require a larger budget if the vendor requires developing the same app for multiple platforms. This increase in cost occurs since developing native apps for different platforms requires knowledge of different software stacks. Native apps developed for Apple iOS will not work on Android OS and vice-versa.

Fig. 2.1 describes the basic architecture of a hybrid mobile app. As mentioned in §5.1, unlike native apps, hybrid companion apps use web technologies for development, and the hybrid app development framework provides *plugins*—a combination of *web APIs* and *native APIs*, for the web content to communicate with *security-sensitive resource* device resources, such as geolocation, microphone, camera, contact list, file, media, storage, and others. These device resources are security-sensitive since they are susceptible to cyberattacks that breach user's privacy [60]. The embedded web browser renders the web content, including local web code, remote web code located on the app's web server, or third-party web code such as ad syndicator scripts or other external JavaScript code. The embedded browser in Android OS is `WebView` [61] and in iOS it is `WKWebview` [62]. We will focus on `WebView` since our research focuses on apps for the Android platform.

2.2 Apache Cordova and Extended Frameworks

Apache Cordova is an open-source hybrid app development framework, initially released in 2009 [63]. Contributors to the Apache Cordova project maintain a set of *core plugins* [64] that allow the app to access basic device resources. Several third-party plugins provide additional support to developers. For instance, `cordova-plugin-chrome-apps-proxy` is a third-party plugin that allows setting a proxy for HTTP or HTTPS, and FTP traffic generated within the app [65].

Over the past years, numerous hybrid app development frameworks such as Phonegap, Ionic

Framework, Monaca, Onsen UI, and Framework7 have been built with Apache Cordova's foundation. These frameworks rely on the Cordova SDK for device resource access but provide developers with numerous UI components, platform-specific styling, and various additional features to make the app appear as close to being native as possible [66]. Since these frameworks are an extension of Apache Cordova, as mentioned in §5.1, we refer to apps built using these frameworks as Cordova-based apps.

2.3 Basic Security Available to Hybrid App Developers

In this section, we discuss the basic security measures readily available for the hybrid app developer. These security measures include *Same-Origin Policy* (SOP) [67], which the embedded web browser enforces automatically, *Content Security Policy* (CSP) [68], which the browser enforces but requires developer configurations, and *Domain allow listing* [69], which the app development framework enforces and requires developer configuration. Domain allow listing is available to all Cordova-based frameworks via a plugin.

Same-Origin Policy (SOP). An *origin* comprises scheme, host, and port number of a URL [67]; two URLs have the same origin if they have the same scheme, host, and port number (if specified). *SOP* is a web security policy enforced by the web browser that restricts how a document or script loaded from one origin can interact with a resource from another origin. In hybrid mobile apps, the embedded browser is responsible for enforcing SOP. As an example, let us assume a user is tricked into visiting `https://yourbank.malicioussite.com` instead of `https://yourbank.com`. On the malicious website, the attacker uses an `iframe` to load the actual bank website `https://yourbank.com`, where the user proceeds to login legitimately. Once the user is logged in, a simple JS (shown in Listing 2.1) on the malicious website can access the DOM elements of `https://yourbank.com` loaded in the `iframe`, such as the user's account balance.

```
1 var balance = frames.bank\_frame.document.getElementByID("accountbalance").value;
```

Listing 2.1: Simple JS to access DOM element of document loaded in iframe

This JS code accesses the `iframe` element (named *bank_frame*), through that the document loaded inside this `iframe`, and through that it accesses the HTML element named *accountbalance*, and gets its value. This JS can be extended to forge requests that can also surreptitiously transfer the user's balance. SOP prevents such cross-site requests from being executed.

Content Security Policy (CSP). *Content Security Policy (CSP)* is a native web browser capability that helps mitigate certain injection attacks, such as XSS, *data-exfiltration*, and *clickjacking* [68]. CSP allows the developer to specify which dynamic resource requests (such as image, script, media, and others) can originate via `WebView`; it also allows the developer to specify the location or domain (web or local) from where to load each resource. CSP allows 15 non-mandatory directives (e.g., `default-src`, `script-src`, `style-src`, etc.[68]) that assist the developer in specifying the allow list of locations/domains. For each directive, a developer may use the wildcard (`*`) to allow loading of the specific resource from *any* location/domain.

The most common way of enforcing CSP in a standard web app is through the HTTP `Content-Security-Policy` response header. However, in hybrid mobile apps, the developer applies a CSP at the *page-level* [70], typically using a `meta` tag.

Domain allow listing. *Domain allow listing* is a security model that controls the app's access to external domains over which the app has no control [69, 70]. Apache Cordova provides a configurable allow list via a plugin, `cordova-plugin-whitelist` [71], to define external domains that an app can access. The current Cordova allow list plugin provides three separate allow lists—*Navigation*, *Intent*, and *Network Request* allow list [69]. Cordova adds the allow list plugin by default to a new project, and the allow list can be configured in the Cordova configuration file, i.e., `config.xml`. New apps, by default, allow access to any URL, i.e., they use a *allow-all* wildcard (`*`).

Figure 2.2: IoT SmartHome Ecosystem

2.4 The Smarthome Ecosystem

As seen in Fig. 2.2, in a smarthome ecosystem, many IoT devices are controlled via companion apps installed on a smartphone. The devices typically range from home appliances such as lights, coffee machines, refrigerators, television to devices that secure a smarthome, such as smart locks, IP cameras, and various other alarm systems. Companion apps for these devices can be *native* or *hybrid*. The remainder of this section provides more background on hybrid apps.

2.5 Related Work

In this section we discuss the related work for our dissertation. Subsections 2.5.1, 2.5.2, and 2.5.3 discuss related work relevant to Chapters 3 and 4. Subsection 2.5.1 discusses works in the literature that protect against malicious third-party JavaScript. Subsection 2.5.2 discusses various works in the literature that define and enforce fine-grained policies for mobile apps. Subsection 2.5.3 discusses works in the field of hybrid mobile app security. Subsection 2.5.4 discusses works in the field that secure IoT companion apps, which is relevant to Chapter 5. Subsection 2.5.5 discusses works in the field of gamification, computer science education in general, and IoT software security education.

2.5.1 Third-party JavaScript Isolation

Numerous solutions in the literature provide protection against malicious third-party JavaScript [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. However, the existing solutions do not capture phone-related attack channels, such as SMS, Wi-Fi, Bluetooth, Contacts, NFC, etc. Due to this, adapting these solutions to the hybrid mobile app environment is not a trivial task and requires significant modification to the existing proposals.

JaTE [20] supports the isolation of third-party JavaScript with labels using `Proxy` in ECMAScript 6. However, JaTE does not provide support for mobile app-based permission. Approaches like Adsafe [18] can be applicable; however, one must extend it with JavaScript bridge APIs, and this approach requires third-party JavaScript to be written in a JavaScript subset, particular to Adsafe. ConScript [24] requires browsers to be modified to enforce security policies. This approach limits the deployment of the protection as it requires modification of the underlying OS.

Adjail [22] and Webjail [27] use `iframes`s to isolate third-party content and provide a mechanism for cross-platform interaction. However, these works cannot protect against attacks on JS plugins/bridges included in hybrid mobile apps since they are accessible by any JavaScript code allowed to load in the app. ScriptProtect [28] automatically strips third-party JavaScript code from the ability to conduct unsafe string-to-code conversions effectively removing the root cause of Client-Side XSS without affecting the legitimate code. JSSignature [29] provides a method to bring digital signatures to third-party JavaScript inclusion where all included JavaScript resources are checked against the integrity, authentication, and non-repudiation risks before the execution. NodeSentry [72] provides a policy infrastructure that allows the combining of common web hardening techniques and measures, common and custom access control policies on interactions between libraries and their environment, including any dependent library. However, none of these solutions protect against unauthorized plugin accesses in hybrid mobile apps.

2.5.2 Fine-grained Policy Enforcement in Mobile Apps

There are various efforts to define and enforce fine-grained policies for mobile apps in general. For example, ConSpec [73] is an automata-based policy specification. ConSpec can specify and enforce both user policies, e.g., users may want to limit the number of SMSs sent from an app, and application contracts, i.e., policies that govern an app's security-relevant behaviors. However, ConSpec targets type-safe byte-code languages only and cannot monitor or enforce policies on JavaScript, a language that is not type-safe. LoPSiL [74] is another policy specification language that can specify and enforce location-dependent security and privacy policies for mobile apps. A sample privacy-based access-control policy in LoPSiL is constraining an app's ability to read location data at specific times.

There are other mechanisms to enforce fine-grained policies for mobile apps; however, they are specific to the Android platform and require the modification of the Android OS. For example, AppGuard [75] is capable of enforcing user-customizable policies on untrusted apps by modifying the apps. AppGuard can enforce fine-grained policies such as the possibility of specifying a set of servers an app is allowed to contact over the Internet. Secure Application INTeraction (Saint) [76] is another access-control system that can enforce both installation time permission granting policies and run time inter-application communication policies. FlaskDroid [77] is another security framework that works simultaneously on both Android's middleware and kernel layers to enforce access-control policies. Apex [78] introduces a user-centric policy specification by extending Android permission with run time constraints with only two parameters: the number of times, and the time of the day.

Another web access monitoring mechanism can monitor all web access via WebView on Android [79]. In addition, this mechanism does not require any modification of the Android Framework and the Linux kernel, and can be introduced by just replacing `WebView` with a modified version. However, it cannot protect against attacks that originate from developer included third-party JavaScript.

2.5.3 Hybrid Mobile Application Security

2.5.3.1 Access Control Systems/Frameworks

Many proposals introduce access control mechanisms for hybrid mobile apps. PhoneWrap [80] enforces fine-grained ticket-based security policies on hybrid mobile apps. These ticket-based policies ensure a bounded number of resource accesses based on the user's interaction with the app. Resource accesses through JavaScript interfaces are wrapped by a library, inspired by the "self-protecting JavaScript" approach [26]. However, PhoneWrap excludes a multi-party scenario and cannot enforce separate policies for different origins as proposed in our work. POWERGATE [39] allows developers to define origin-based access control, however, POWERGATE only protects native objects and relies on the web-browser to protect DOM objects. Also, its implementation requires modification of the underlying OS. Another work introduces a context-aware permission control system for hybrid mobile apps [81]. This system aims to enforce information flow policies to prevent potential data leakage.

Draco [32] provides a declarative policy language for developers to define fine-grained access control policies for multiple origins, for web code running on Android in-app browsers. It also introduces the Draco Runtime System (DRS) to enforce these policies at runtime. Another fine-grained access control mechanism for Android hybrid mobile apps implements *frame*-level access control [33]. RestrictedPath [34] allows developers to define intended API paths of their apps and subsequently monitors all API invocations. The monitoring will determine whether an app deviates from its intended path [34], thus enforcing access-control. MinPerm [17] automatically identifies over-privileged permissions by comparing permissions declared by the developer and permissions actually required by the app. However, all these approaches are specific to Android and thus, require the modification of the Android base system.

Georgiev et al. introduce the term *fracking* for the generic class of vulnerabilities that allow untrusted web content to access device resources [37]. They propose NOFRAK, an access control mechanism that enforces a security policy, "NoBridge"—an app can load third-party

content, but this content cannot access device resources. This approach is a highly coarse-grained mechanism as it only allows/disallows a third-party JS to access device resources. It cannot enforce fine-grained policies, such as, allow limited access (only `read` access), or put a bound on the number of accesses. AlJarrah et al. propose an access-control mechanism that restricts access to only required device resources per page, to minimize the attack surface [16]. However, this solution is only applicable to multi-page hybrid mobile apps. The same researchers also propose a behavior-based approach to generating fine-grained security configurations to implement the least privilege principle automatically [35]. In another work, CordovaConfig [82], they implement a web-based tool prototype that provides automated interactive support for configuring hybrid apps. Kudo et al. [38] introduce a novel attack technique termed as *app-repackaging*, where an attacker repackages hybrid apps with malicious code intended to steal sensitive user data stealthily. They introduce a run-time access control mechanism to restrict access to the device resources. However, all these approaches modify the underlying Cordova library to implement the solution.

Yang et al. [83] identify a new security issue in `postMessage` in hybrid mobile apps. The work demonstrates that origin information of a message in is not respected or even lost during the message delivery. This issue allows adversaries to inject malicious code into `WebView` to passively monitor messages. These messages may contain sensitive information, or actively send messages to arbitrary receivers and access their internal functionalities and data. The authors term this issue as *Origin-Stripping Vulnerability* (OSV) and develop a tool called OSV-Hunter to detect such vulnerabilities. They also develop a defense tool to mitigate OSV by implementing three new postMessage APIs, called OSV-Free. However, OSV-Free cannot protect against fracking attacks or allow the developer to enforce principal based fine-grained policies on device resources.

2.5.3.2 Detecting & Preventing Code-Injection

Several solutions focus on detecting code-injection attacks in hybrid mobile apps. Jin et al. introduce the possibility of code-injection attacks in hybrid mobile apps through non-web

channels, such as SMS, Contact List, Calendar, NFC, camera and even Wi-FI SSID, that are specific to smartphones [1]. DroidCIA [84] extends the previously mentioned work, i.e., [1] to introduce a new code-injection channel, where a malicious script can be injected by using the HTML5 `textbox` element along with `document.getElementByID("TagID").value` [84]. Xiao et al. introduce a new type of code injection attack that encodes the injected JavaScript code in a human-unreadable format [85]. The authors use machine learning algorithms to detect vulnerable apps and also suggest an improved access control model that uses a combination of page-based and frame-based techniques. Yan et al. present a new deep learning network, Hybrid Deep Learning Network (HDLN), and use it to detect code-injection attacks [86].

Another work proposes an approach to detecting code-injection in hybrid mobile apps by monitoring the execution of apps, and generating runtime-behavior state machine is based on the execution contexts. Any deviation from the original behavior state machines aid in detecting the code-injection [87]. *SCANCIF* [88] is a static analysis tool identifying sensitive plugin APIs based on tags that can inject malicious code. The work also analyzes information flow based on modeling contexts of callback functions passed in function calls. BRIDGETAINT [89] is a novel bi-directional dynamic taint tracking method that can detect bridge security issues in hybrid apps. BRIDGEINSPECTOR [89] is a tool based on BRIDGETAINT that detects cross-language privacy leaks and code-injection attacks in hybrid apps.

In summary, all thes works mentioned above detect and prevent code-injection attacks that can execute malicious code at runtime. However, hybrid app developers can prevent such code-injection attacks by disallowing inline scripts in CSP.

2.5.3.3 Security Analysis and Surveys

There are a few studies that provide an overview of security mechanisms and analyze the vulnerabilities in hybrid mobile apps. In [90], the authors reveal that 28% of one million web-based mobile apps have at least one vulnerability. If exploited, these vulnerabilities

can cause serious cyber-attacks. [91] studies over a thousand Cordova apps downloaded from Google Play and gives a statistical overview of the adoption of Cordova security best practices and mechanisms, such as usage of allow list or the occurrence of `eval()`, among others. Another study of 2111 hybrid mobile apps analyzes configurations and permissions usage patterns [36]. In that work, the authors provide systematization of hybrid mobile apps configuration model. It shows the evidence of configuration misuse and tendency of developers to use default settings and possible reasons for misconfigurations.

In [92], the authors summarize the statistics of the prevalece of hybrid apps, most widely used cross-platform tools, based on the analysis of around 15,000 hybrid apps. BridgeScope [93], investigates JavaScript bridge security issues, such as evading security checks in WebView event handlers, in Android hybrid apps. HybriDroid [94], a static analysis framework for Android hybrid apps, investigates bugs originating from the interoperability of Android Java and JavaScript in Android hybrid mobile apps. Hybrid-scanner [95] is another tool that tracks and analyzes the internal behavior of hybrid mobile apps. Using Hybrid-scanner, the authors found that almost 40% of security-sensitive APIs in hybrid mobile apps are invoked by third-party libraries, e.g., advertisement libraries. Apart from revealing numerous security issues in hybrid mobile apps, none of the works implement any defense solution.

A comprehensive survey [96] assesses the cross-platform mobile app development academic body of knowledge with a particular emphasis on core concepts that include user experience, device features, performance, and security. Their findings illustrate that the state of research demands for empirical verification of an array of unbacked claims, and that a particular focus on qualitative user-oriented research is essential.

2.5.4 IoT companion mobile app security

Not a lot of work in IoT security focuses on exploitability of IoT devices through their companion mobile apps or to make them more secure. A static source code analysis of 499 SmartThings apps (*SmartApps*) and 132 device handlers, with the help of carefully crafted

test cases reveals that 55% of these apps are overprivileged [43]. The study reveals that once installed, a SmartApp is granted full access to a device even if it requires only limited access to the device. The study also reveals that the SmartThings event subsystem, which devices use to communicate asynchronously with SmartApps via events, does not sufficiently protect events that carry sensitive information such as lock codes. The researchers use four proof-of-concept attacks to secretly planted door lock codes, steal existing door lock codes, disable vacation mode of the home, and induce a fake fire alarm [43]. IoTFUZZER [48] is a novel fuzzing framework that aims at finding memory corruption vulnerabilities in IoT devices without access to their firmware images. IoTFUZZER was evaluated on 17 real-world IoT devices running on different protocols, and successfully identified 15 memory corruption vulnerabilities in these devices (including 8 previously unknown ones). IotSan [97], another novel practical system uses model checking as a building block to reveal interaction-level flaws by identifying events that can lead the system to unsafe states. IotSan automatically translates IoT apps into a format amenable to model checking. An attribution mechanism helps in identifying problematic and potentially malicious apps. Evaluation of IotSan on the Samsung SmartThings platform reveals that after testing 76 manually configured systems, IotSan detects 147 vulnerabilities. SOTERIA [45], is another static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. Evaluation of SOTERIA on 65 SmartThings market apps through 35 properties and find nine individual apps violate ten properties. Evaluating SOTERIA on MALIOT, a novel open-source test suite containing 17 apps, revealed 20 unique violations [45]. Another work that analyzes communication between IoT devices and their companion mobile apps reveals that the communication between an IoT device and its app is often not properly encrypted and authenticated and these issues enable the construction of exploits to remotely control the devices [47]. To confirm the vulnerabilities found, the paper also presents exploits against five popular IoT devices from Amazon by using a combination of static and dynamic analyses. The work

also discusses defense strategies that developers can adapt to address the lessons from our work [47]. Another work presents a platform that does cross analysis of IoT companion mobile apps to infer components that are reused across multiple devices and use this data to discover vulnerability in IoT devices [46]. Using a suite of program analysis techniques included in the platform, a large-scale analysis is performed on 4,700 devices. The study highlights the sharing of vulnerable components across the smart home IoT devices (e.g., shared vulnerable protocol, backend services, device rebranding), and leads to the discovery of 324 devices from 73 different vendors that are likely to be vulnerable to a set of security issues [46]. Another work proposes a modeling methodology to study home-based IoT devices and evaluate their security based on component analysis that includes the IoT device, the companion mobile app, the cloud endpoints, and the associated communication channels [44]. The study systematizes the research literature for home-based IoT devices to understand attack techniques, proposed mitigation, and stakeholder responsibilities. The study also evaluates the systematization on 45 home-based IoT devices that are available on the market today and provide an overview of their security properties across the IoT components.The study also establishes a portal where researchers, vendors, and power-users can contribute to new device evaluations and to reproduce the results using the published dataset and proposed methodology [44].

2.5.5 Gamification and Education

2.5.5.1 Gamification

Gamification is not a new concept in cybersecurity education and training and has been applied in different areas of the field [98, 99, 100, 101]. Numerous works establish the benefits of gamification in making cybersecurity education more engaging and enjoyable (cf., [102]). However, not a lot of works focus on gamification of activities that teach and assess skills required to reverse engineer and analyze IoT firmware. Ashgar et al. discuss an approach to teach reverse engineering in a classroom environment but their focus is on reversing the app code for a mobile app [103]. With Criminal Investigations, we focus on reverse engineering

and analysis of IoT firmware, which is very different than reverse engineering a mobile app.

Gamified activities have been shown to increase student engagement and learning [104]. An example of such a gamified activity is a set of "wargames" created by the OverTheWire community [105]. Watson et al. [106] found that once students reach a certain level of engagement into a gamified activity, they are likely to continue to optional, ungraded levels.

We utilize game design principles to ensure the delivery of experiences that are meaningful and engaging [107]. A key aspect of this approach is the idea of games posing a challenge to the players; for example, the need to overcome an obstacle or manage a key resource to successfully achieve the game's objectives [108]. Furthermore, well-designed games often tell compelling stories and enable the audience to be active participants in an interactive experience [109, 108, 107]. Games typically also include sophisticated rule systems and rewards mechanisms, designed to promote specific activities and discourage or prevent others [110, 107]. We leverage these characteristics of game design to create a framework that enables the delivery of engaging experiences. Players are presented with scenarios built around specific learning objectives, supported by hands-on activities conducted in an interactive environment [109, 111], to emphasize key concepts or essential skills.

2.5.5.2 Education

Over the last several years, there has been a large body of research on pedagogical strategies to help students develop higher-order thinking skills [112], to improve student engagement and to support inclusivity. A central idea that helps achieve these goals is active learning [113, 114, 115]. Gamification is one approach to improve student engagement and increase motivation [104, 106]. Manifestations of active learning may be found in team-based learning [116, 117], the Flipped Classroom [118, 119, 120, 121], and Process-Oriented Guided Inquiry Learning (POGIL) [122, 123, 124, 125, 126, 127, 128].

2.5.6 Hybrid Mobile App Security Education

Related work in hybrid mobile app security education (such as advanced attacks and defenses) is sparse at the time of writing. The closest related work to ours is a webinar that discusses security issues in hybrid apps, such as complexity of the JavaScript language, i.e., not type-safe, highly dynamic, etc., inclusion of large third-party libraries and modules and complexities in cross-language analysis [129].

2.5.6.1 IoT Software Security Education

The closest works to ours in IoT security are two advanced IoT security training course/-workshops that include firmware—extraction, emulation and analysis, and building exploits for ARM and MIPS architectures [130, 131]. Another training course/workshop covers broader topics, such as secure architecture, infrastructure, policies, mobile and cloud vulnerabilities and briefly touches upon firmware analysis [132]. However, we plan to incorporate activities in Criminal Investigations that teach how to use *Address Sanitizer* [133] and American Fuzzy Lop (AFL) [134] to identify vulnerabilities in IoT firmware and writing advanced exploits for ARM and x86 that can bypass memory protections. Other vendors such as Udemy [135] and EdX [136] explore limited introductory IoT security topics [137, 138], such as identifying and analyzing IoT security and privacy risks, understanding conceptual designs for secure hardware and software, knowledge of security architectures, etc. However, these do not address IoT firmware security and neither are they gamified.

Chothia introduces a classroom course that focuses on basic end-to-end penetration testing techniques for IoT devices and includes a basic IoT firmware security (simple buffer overflows) module [139]. In contrast, our course will discuss firmware analysis and security in depth, including firmware extraction, reversing, analysis and fuzzing, using tools such as AddressSanitizer to identify memory-corruption vulnerabilities and writing advanced exploits that can bypass memory-level protections.

CHAPTER 3: HYBRIDGUARD: A PRINCIPAL-BASED PERMISSION AND FINE-GRAINED POLICY ENFORCEMENT FRAMEWORK FOR HYBRID MOBILE APPLICATIONS[1]

3.1 Introduction

In this work, we present HybridGuard [55], a novel policy enforcement framework based on *inlined reference monitors (IRMs)* [140] that can enforce principal-based, stateful policies, on multiple origins *without* modifying the hybrid frameworks or mobile platforms. In HybridGuard, hybrid app developers can specify principal-based permissions, and define fine-grained, and stateful policies that can mitigate a significant class of attacks caused by potentially malicious JavaScript code included from third-party domains, including ads running inside the app. HybridGuard also provides template policy patterns and allows app developers to specify fine-grained policies for multiple principals. HybridGuard is implemented in JavaScript; therefore, it can be easily adapted for other hybrid frameworks or mobile platforms without modification of these frameworks or platforms. We present attack scenarios and report experimental results to demonstrate how HybridGuard can thwart attacks against hybrid mobile apps. The main contributions of our work include:

- A robust IRM framework for hybrid mobile app developers to specify and enforce useful security policies to protect the users from potential cyber-attacks.
- A novel principal-based permission access control and fine-grained security policy specification for hybrid mobile apps.
- A wide-range of security policy patterns that can be enforced in hybrid mobile apps to

[1]This chapter includes previously published ([55]) joint work with Phu Phung, Rahul Rachapalli, and Meera Sridhar

prevent real-world attacks.

- A small-scale experimental evaluation of our proposed framework on Android and iOS platforms.

Roadmap. The rest of this chapter is organized as follows. Section 3.2 discusses our threat model for HybridGuard and presents running attack scenarios for this chapter and Chapter 4. Section 3.3 provides an overview of HybridGuard and the different types of policies that HybridGuard can implement. Section 3.4 provides details about HybridGuard's implementation. Section 3.5 provides details about the policy classes that HybridGuard can enforce. Section 3.6 provides details on HybridGuard's evaluation, and §3.7 concludes.

3.2 Threat Model and Running Examples

In this work, we consider scenarios where hybrid mobile apps are developed by trusted and legitimate developers and therefore, are trusted by users. We consider two threat models in this work:

- **Threat Model 1:** This threat model is used in our HybridGuard works (Chapters 3 and 4) and assumes that a hybrid mobile app developer includes CSP in the app to protect against code injection attacks. The in-scope threats originate from third-party JavaScript code included from a source allow listed in the CSP. The third-party JavaScript code could be 1. benign but under the control of an attacker through web application attacks, such as SQL injection, or a network attack on the third-party server; 2. malicious by intentions; it lures developers by its appealing functionalities.

- **Threat Model 2:** This threat model is used in our work on securing IoT companion mobile apps (Chapter 5) and assumes that IoT companion mobile apps are vulnerable to web-attacks, such as code-injection attacks, XSS, CSRF, SQLi, Sensitive Data Exposure and Broken Authentication & Session Management.

3.2.1 Running Attack Scenarios

Abusing device resources. Consider a hybrid mobile app that requires access to SMS and Contacts. By default, after a user grants the required permissions (could be at installation or run time), any JS code running inside the application has access to these device resources [141]. As seen in Figure 3.1(a), if a allow listed third-party JS code is infected with malicious code controlled by an attacker, the malicious code can access all the granted resources. For example, the malicious code can send bulk spam SMS messages to the user's contacts or random numbers.

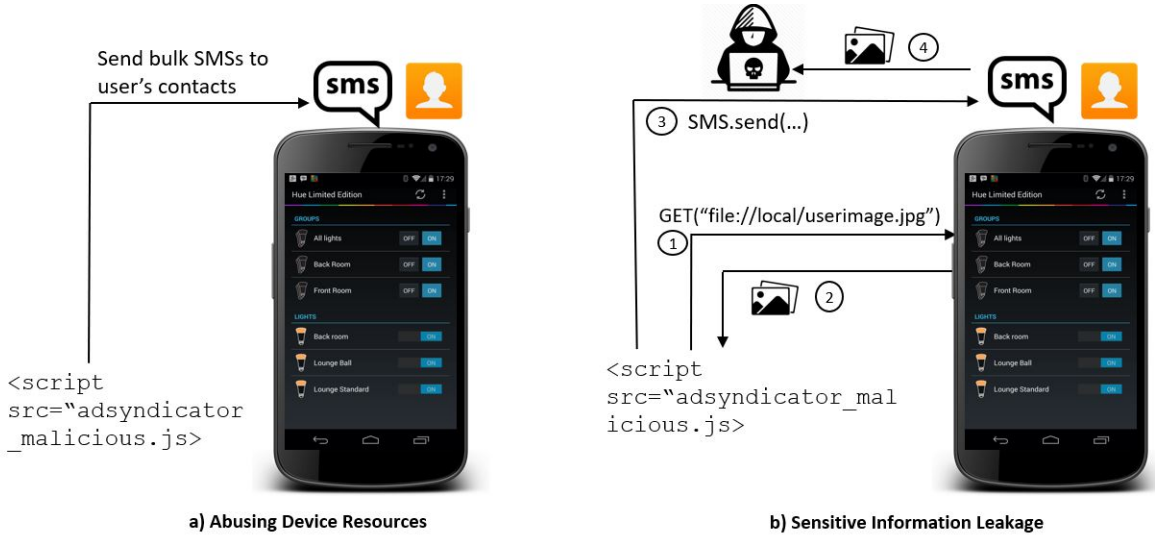

Figure 3.1: Abusing Device Resources & Sensitive Information Leakage

Malvertising and sensitive information leakage. Most free apps display in-app ads [142] to generate revenue through clicks and referrals. Due to CSP, a developer needs to explicitly add the ad network's URL to the allow list to display ads in the app. These ad services have an extensive screening process of supplied ads; however, the process is not airtight [143] since there have been many malvertising [144] incidents in the past [145, 146, 147]. Malicious ads have made to users' devices by either slipping through the screening process or by compromising the ad network. In a hybrid mobile app, the ads are fetched by including an ad network provided JS in the app code, usually the landing HTML page (home page) of the

app. Figure 3.1(b), if this JS is malicious, it can read the user's sensitive information that may be available on the host page, such as media, sensitive files or other personal information. Although, CSP disallows information to be sent to any external domain not in the allow list, malicious JS code in an app with access to SMS and Email APIs, can use these channels to exfiltrate the stolen sensitive information.

Overusage of Resources. As seen in Figure 3.1(a), a rogue third-party JavaScript with unlimited access to granted device resources, such as geolocation, can constantly monitor the current location of a user in real-time by hooking the `navigator.geolocation.watchPosition()` API and exfiltrating this data to the attacker. This attack scenario does not only breach the privacy of the user but also puts the physical safety of the user at risk.

UI redress attacks. UI redress attacks are also known as clickjacking on the web or touchjacking/tapjacking on smartphones [148, 149]. Malicious code can manipulate the DOM of the host page, including the creation of new elements or the modification of existing ones. As seen in Figure 3.2(b), leveraging this ability, malicious JS code in a hybrid app can launch such attacks by creating an invisible interface, such as an invisible `iframe` on top of the app interface. Here, the attacker tricks a user into tapping a button or link on another page loaded in the `iframe`, assisting the attacker in 'hijacking' clicks to perform actions on behalf of the user, on the page loaded into the `iframe`. This scenario can also lead to drive-by-download attacks.

3.3 Overview

Figure 3.3 gives an overview of the entire toolchain. In the first step, the APK is reverse engineered using `apktool` [150] to obtain the app source code. In the second step, the *rewriter* injects HybridGuard into the main HTML file of the app, usually `index.html`, and includes both HybridGuard (JS file) and the policy specification file (JSON file) in the `www` directory of the unpacked APK. Finally, the original files of APK combined with HybridGuard and the policy file are repacked using `apktool` to produce a safe APK. HybridGuard enforces the

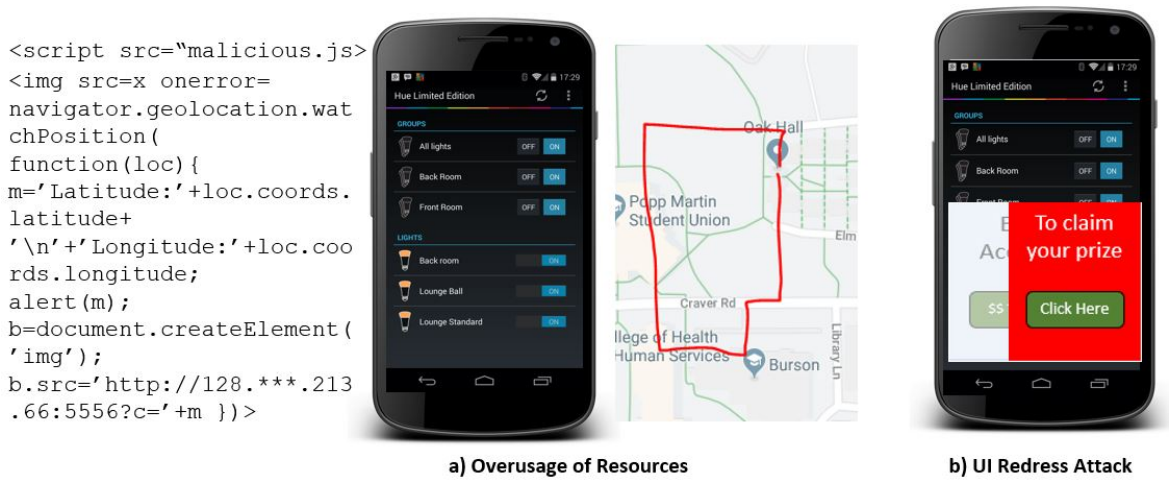

Figure 3.2: Overusage of Resources & UI Redress attacks

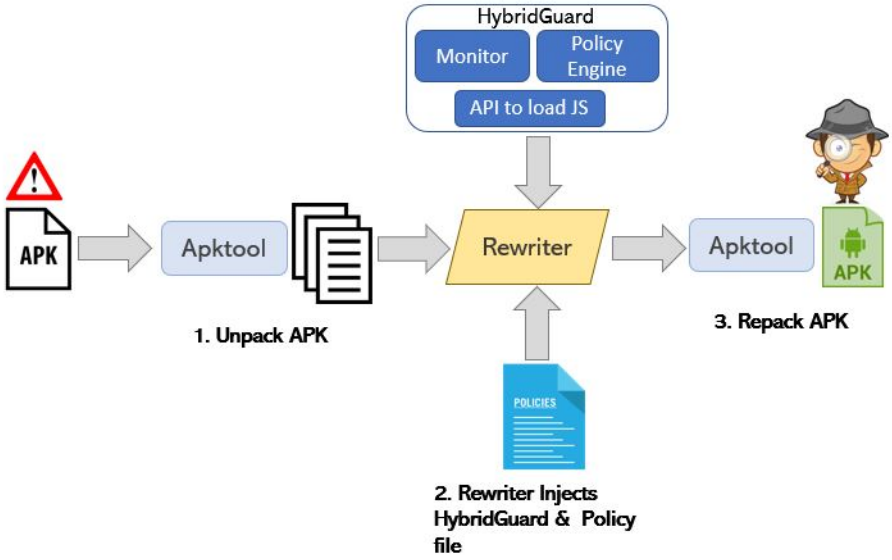

Figure 3.3: HybridGuard Overview

policies at app runtime.

Note, in Figure 3.3 HybridGuard contains various components that include:

1. an *API* to load the JS files that the developer requires to include in the app (More Details in §3.3.1);

2. a *monitor* that mediates all security relevant events that originate from the include JS (More Details in §3.3.2);

3. a *policy engine* that is consulted by the monitor to check the policy to allow/disallow

the event, and the policies specified by the developer (More Details in §3.3.3).

The rest of Section 3.3 provides an overview of the three components mentioned above, and the policy enforcement approach. A detailed description of our implementation is presented in Section 3.4.

3.3.1 An API to load JavaScript Code

As seen in Figure 3.3, HybridGuard provides the developer with a new API to load JS (`*.js`) files. This interface allows hybrid mobile app developers to assign a principal to each JS file that needs to be included in the app, which is the basis for policy definition and enforcement. Instead of using the traditional HTML <script> element to include JS, the developer uses HybridGuard's API to include the required JS into the app code. The developer can also include local JS files under a named principal (trusted principal) to define fine-grained security policies. An important goal of HybridGuard is to ensure attribution of the JS code with the assigned principal while the JS code is in execution.

JS code is executed in the sequence of inclusion in the main page of the app, i.e., in order of appearance and "run-to-completion" [151]. However, at runtime, additional JS code can be generated and executed on the fly. This code may be either generated dynamically by different JS code or can be code embedded into event handlers. Therefore, HybridGuard must monitor the principals even during context changes due to dynamic code generation and event triggers. In hybrid mobile apps, there can be a trusted principal (local code) and multiple third-party principals. We use a local principal stack to track the various principals at runtime. Whenever JS code is executed, its principal is pushed onto the stack. When the code terminates, the principal is popped from the stack. HybridGuard explicitly tracks dynamic code generation and event handlers, and executes them under the same principal that generates the code. HybridGuard ensures that the code is attributed to the correct principal and the appropriate policy is enforced. HybridGuard keeps track of the principal at runtime to ensure appropriate enforcement of the defined policies.

3.3.2 API Mediation

HybridGuard's monitor component (Figure 3.3) intercepts security-relevant API calls including access to the device resources and the DOM elements. Security-relevant API calls originating from JS code (local or third-party) are marked with the principal assigned at load-time, and intercepted by the monitor to verify access based on the defined policies. HybridGuard intercepts these API calls by wrapping them and checking the policy to determine if a call is allowed or not. The monitor refers to the policy engine to decide whether to grant or deny the API call based on the specified policies.

HybridGuard's API to load JS, monitor and policy engine are implemented in one single JS file. This file is included using a `<script>` tag in the main page, right after the framework's plugin APIs JS file, such as `cordova.js` for Apache Cordova framework. When executed, HybridGuard mediates all guarded "plugin" APIs and DOM APIs. HybridGuard loads the required JS code, including both local and remote code. This code also includes third-party JS code that could be potentially malicious. Loading the JS code through HybridGuard's interface guarantees that this code cannot access any resource via the original APIs but only via the mediated/wrapped APIs. This allows HybridGuard to control code execution based on defined policies.

3.3.3 Principal-based, Fine-grained Security Policies

The current permission model on both Android and iOS is too coarse-grained and only allows/disallows access to any device resource. Moreover, once a permission is granted, there is no control over how the app uses that permission. HybridGuard allows developers to define fine-grained security policies for multiple principals/third-parties using the policy specification file (Figure 3.3 Step 3). Based on the specified policies, HybridGuard controls access to device resources and other security-relevant APIs. HybridGuard is capable of enforcing any access control or safety policy that can be expressed as a security automaton [152], as shown in Figure 3.4.

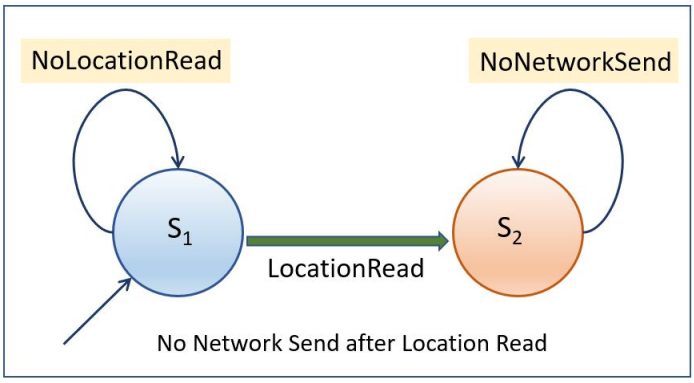

Figure 3.4: A simple security policy expressed as a security automaton

For policy design and specification, we use JavaScript Object Notation (JSON) [153] since the *key-value* pair structure and lightweightedness of the language makes it a pertinent choice. As shown in Listing 3.1, policies are specified in a single file (`.json`), and supplied to the policy engine. Using HybridGuard a developer can enforce principal-based permissions at the API level, i.e., a level deeper into the device resource. HybridGuard complements the existing OS permission model by introducing access qualifiers such as `read`, `write` and `create`, instead of just allow/disallow. Besides these access qualifiers, HybridGuard allows the developer to create an *allow list* to limit resource access to certain predefined principals. Developers can also set principal-based *access-bounds* to limit access to a specific device resource based on a set bound. As an example, Listing 3.1 line 12 depicts allow list usage, where the value of the key ('`numbers`') is a allow list of contact numbers an app is allowed to send an SMS to.

```
1 {"resources": [{
2    "name": "sms",
3    "permissions": [{
4     "principal": "local",
5     "read": "true",
6     "write": "true"
7    },
8    {
9     "principal": "trusted.com",
```

```
10      "read": "true"
11     }]
12     "numbers": ["1234567890", "5682241205", "2254813544"]
13  //...
14  }
```

Listing 3.1: Principal-based policy specification

HybridGuard is capable of enforcing fine-grained stateful policies that can be further categorized as resource-bounds, allow list, history-based and custom policies.

3.4 Implementation

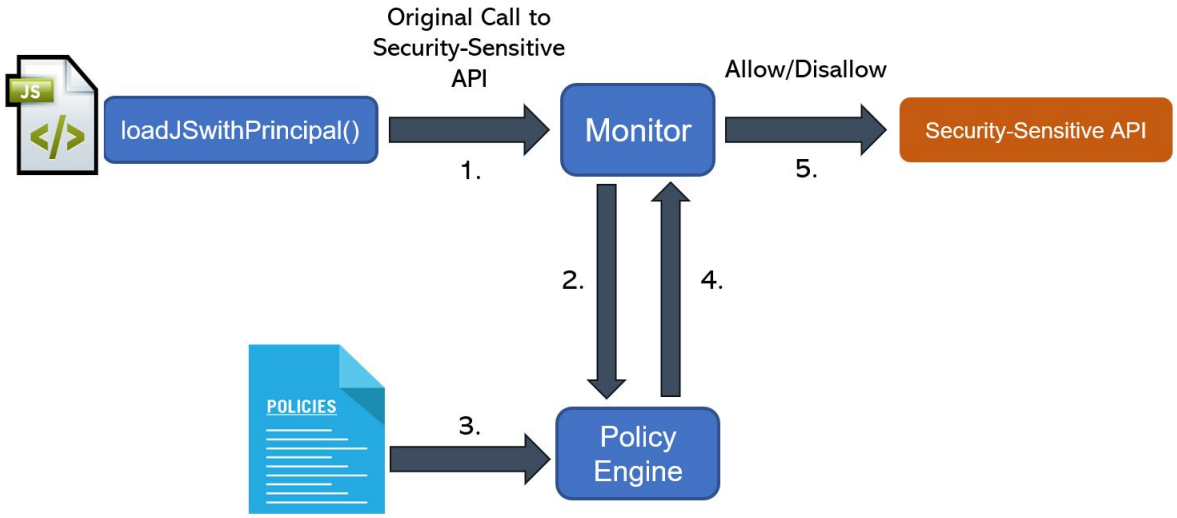

Figure 3.5: HybridGuard's components and policy enforcement

Figure 3.5 depicts different components of HybridGuard and their interactions with each other to enforce the specified policies. As seen in Figure 3.5, a JS (`.js` file) that requires monitoring, is included in the app using the `loadJSwithPrincipal()` API provided by HybridGuard, and as singed a named principal. Subsequently, as seen in Figure 3.5 Step 1, any invocation of security-sensitive APIs, originating from the monitored JS code (marked with the assigned principal), is mediated by the monitor. In Step 2, the monitor consults the policy engine to ensure that the specified security policies are satisfied. In Step 3, the

policy engine consults the policy specification file to identify if a certain principal is allowed to access the security-sensitive API. If there is no policy violation, in step 4, the policy engine notifies the monitor to allow access. In step 5, the monitor allows the invocation and forwards the call security-sensitive API. Now, we provide technical details of each component and the policy enforcement mechanism.

3.4.1 Custom Script Execution with Principal

The origin of JS code in hybrid mobile apps is not propagated, therefore, the app developer cannot enforce policies based on the real origin of the included JS [32]. To overcome this obstacle, we design a new JS API `loadJSwithPrincipal(p, url)` (included in HybridGuard) that replaces the conventional method of script inclusion. The app developer can use this API to load and execute a JS file, local or remote, described in the `url` argument under a principal `p`. For example, instead of using `<script src="http://example.com/ad.js"></script>` to load the external JS from `example.com`, the app developer uses `loadJSwithPrincipal(..)` to include the code under a named principal "`example.com`" as `loadJSwithPrincipal("example.com", "http://example.com/ad.js");`.

We adapt a previous approach to implement the `loadJSwithPrincipal` API [19]. Different from the previous approach, we use Cross-Origin Resource Sharing (CORS) [154] request to retrieve the content of the JS file in a string. We can retrieve both local or cross-domain remote files in the same way using CORS request, using the `XMLHttpRequest` object. Then, we create a new `Function` object with the retrieved JS content. Then, we push the assigned principal `p` to a local protected stack (implemented as an array), execute the function, and pop the principal off the stack after the execution is complete.

3.4.2 JavaScript APIs Mediation

An essential feature of HybridGuard is the capability to monitor JS APIs, which includes DOM APIs and JS plugin APIs. This is achieved by wrapping APIs that require monitoring, so that invocations of the original APIs are mediated by the wrapper APIs. The wrapper

APIs are part of the monitoring code, and invoke the policy engine to determine whether to allow/disallow the invocation. This approach is inherited from prior work [26], and depicted in Fig. 3.5. We have advanced the previous work by implementing mediation for JS plugin APIs and principal-based permission access control, which does not exist in the state-of-the-art JavaScript security solutions.

```
1  function(){
2      var original = sms.send;
3      sms.send = function () {
4          if(PolicyCheck(getTopofPrincipalStack(),"sms","send",arguments)){
5              original.apply(this, arguments);
6          }else{
7              throw new Error('sms.send is disallowed');
8          }
9      }
10 }();
```

Listing 3.2: Illustration of mediation of API sms.send within an anonymous function.

One challenge in this approach is to ensure complete mediation of security relevant events, i.e., ensure that the monitored JS code cannot access the guarded APIs directly but only through the monitor. For DOM APIs, this is achieved by capturing all possible aliases of the guarded APIs through their prototype inheritance chain [155]. There have been several known JS vulnerabilities that can be exploited in JS interception approaches [24, 156]. We apply the secure wrapper implementation [156] in the literature to ensure that our monitor code is tamper-proof from known JS vulnerabilities and potentially malicious code.

For JS plugin APIs, there can be several different APIs provided by various plugins to access a device resource. Since the plugins are included in the app by the developer, he/she knows the specific APIs to intercept and enforce policies on. Each JS plugin API typically uses an internal function call to interact with the native API. For example, in Cordova, `exec` is the internal function to interact with Java API. To ensure that JS code loaded by our framework cannot interact with the native APIs directly, we also intercept this internal function.

3.4.2.1 Principal Propagation in Event Handlers and Dynamic Code Generation

Similar to native mobile apps, hybrid mobile apps heavily rely on events such as user's touch to trigger code execution. HybridGuard captures and intercepts these event channels, such as `addEventListener` and `attachEvent`, to wrap the handler functions so that when the event is fired, e.g., a button is touched, the handler function is executed under the same principal as the parent code. This allows to enforce the same policy for that handler function as well. This approach is illustrated in Listing 3.3

```
var eventguard = function(args, proceed) {
    var principal = getTopofPrincipalStack();
    var listener = args[1]; //the handler function
    args[1] = function(){
       //wrap the handler function to be executed
      //with the same principal
       return execWithPrincipal(principal,listener);
    };
    return proceed();//register the event handler
  };
intercept(Element.prototype, 'addEventListener', eventguard);
intercept(Node.prototype, 'addEventListener', eventguard);
```

Listing 3.3: Principal Tracking for event handler

The same approach is applied for code generation on the fly through DOM APIs, such as `document.write`, `Node.insertBefore(..)`. Since inline JavaScript code in HTML is not allowed by CSP by default, we only need to ensure that new script nodes created by existing JavaScript will be executed under the same principal as the script that created it.

3.4.3 Policy Management and Enforcement

As illustrated in Fig. 3.5, an invocation to a guarded API will be dispatched together with its principal to the corresponding monitor. The monitor then consults the policy manager; based on policy definition, the policy manager will decide whether to proceed the invocation. As briefly outlined in the previous section, our framework supports principal-based permission and stateful policies. We design and implement the policy specification for HybridGuard as follows.

```
{"resources": [{
   "name": "contacts",
   "permissions": [{
    "principal": "local",
    "read": "true",
    "write": "true"
   },
   {
    "principal": "trusted.com",
    "read": "true"
   }]
//...
}
```

Listing 3.4: A principal-based permission example

3.4.3.1 Principal-based Permission

We use JavaScript Object Notation (JSON) to specify principal-based permission for the device resource access (including DOM and JavaScript bridge APIs) by any JavaScript code running inside the app. The device resources are specified as an array of objects inside the JSON file, and each device resource object has an array of permission objects of its own. The permissions to access the device resources are defined by a principal. For each resource, the app developer can specify which principal is allowed to access (read or write) which APIs. For instance, Listing 3.4 illustrates an example of principal-based permission that allows the local code (loaded with principal "`local`") to read and write on the contact resource, while allows JavaScript code from "`trusted.com`" read-only permission. JavaScript code loaded with other principals is denied access to this resource by default in this example.

This JSON specification can be defined and stored in a local variable within the monitor code, however, to separate policy definition from the code, we store it in a local JSON file and load it using `XMLHttpRequest` to perform principal-based permission check for the policy manager.

```
var principal_permission_check = function(principal,resource,action){
    if(json_permission==undefined)
       json_permission = get_json_permission();
    for(i=0;i<json_permission.resources.length;i++){
     if(json_permission.resources[i].name == resource.toString()){
      for(j=0;j<json_permission.resources[i].permissions.length;j++){
         if(json_permission.resources[i].permissions[j].principal == principal){   if(json_permission.
      resources[i].permissions[j].[action] == true){
                    return true;
                }
                else{
                    return false;
                }
        //....
    return;
}
```

Listing 3.5: Function to check permissions (Partial Code)

3.4.3.2 Custom and Fine-grained Security Policy Enforcement

The principal-based permission model can enforce policies to allow or disallow access to a resource; however, it cannot capture and prevent potential malicious actions, such as sensitive information leakage or UI attacks, as we discussed in the motivating attack examples. In addition to the principal-based permission check, HybridGuard also allows the developer to define custom and fine-grained policies such as allow list specification, stateful, and history-based policies. These policies can also be generalized in a specification; however, we leave this for future work. In this framework, these custom policies can be defined in JavaScript code. For example, to prevent a potential information leakage, the developer can define a policy that "after a principal reads the contact list (assume that the principal is allowed to read the contact list in principal-based permission), it is not allowed to send any SMS". This policy is illustrated in Listing 3.6. We note that this policy is also *principal-based*: the principal violating the aforementioned example policy is denied to send SMS, but other principals such the first-party code can still be allowed to send SMS.

3.4.4 Security Analysis

As discussed earlier, potential code injections and information leakage attacks by the web channels can be eliminated by the standard CSP in hybrid mobile apps. HybridGuard provides an extra layer of protection on JavaScript code that is allowed by CSP. As required by default CSP, each JavaScript code must be defined in a `.js` file, either first-party or third-party code. HybridGuard provides a new JavaScript API to obtain the content of these `.js` files and executes them under a principal. This requires HybridGuard's code to run before other first-party or third-party code in the app so that it has the highest priority to control the behavior of the loaded code. As described in the implementation, HybridGuard's code and security states are protected within an anonymous function, which is inaccessible to external code. Access to `JSON` policy specification file is prohibited from unauthorized principals, enforced by the monitor. Therefore, the integrity of HybridGuard is guaranteed. Adapting known techniques from prior work [156], HybridGuard ensures the *complete mediation* of JavaScript web APIs by systematically discovering and mediating all their possible aliases and channels generating JavaScript code on the fly. For JavaScript bridge APIs provided by hybrid frameworks, we have to manually identify the possible channels for each API to ensure it is completed wrapped. Since HybridGuard can control the behavior of the loaded code, any unauthorized access can be detected and prevented.

3.5 Fine-Grained Security Policies

As discussed earlier, in addition to principal-based permission specification, HybridGuard allows hybrid app developers to define more fine-grained security policies. Implemented as a in-lined reference monitor framework, HybridGuard supports fine-grained security policies that satisfy safety property of execution, i.e., prevent bad things from happening. The app developer knows the functionality of the app, which resources will request permission from the user, and even the confidential information in the webpage of the hybrid app. When including third-party code, the developer can, therefore, define permission for each party

through a principal. In this section, we present some useful policy patterns that the hybrid app developer can leverage to protect the end-users.

```
var contact_read_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"contacts", "read"))
        return; //no permission for this principal
    toggle(contact_read);// update the contact read history
    if(!bound_check(p, "contact", "read") return;
    return proceed();//allow the invocation
  };
var sms_send_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"sms", "send"))
        return; //no permission for this principal
    if(contact_read) return;
    if(!bound_check(p, "sms", "send") return;
    if (!allowlist_check(p, "sms", "send", args[1])) return;
    return proceed();//allow the invocation
  };
intercept(sms, 'send', sms_send_policy);
intercept(navigator.contacts, 'find', contact_read_policy);
```

Listing 3.6: Example of "no SMS send after reading contact list"

Resource Bounds Policy. In past, mobile apps in general have been susceptible to overuse and abuse of resources, and during our experiments we encountered numerous apps that request more permissions than required. Consider an app that sends greetings to your contacts on their birthdays. Disallowing access to a resource (`Contacts` in this case) will break the app's functionality and is not a feasible policy. In such a scenario, the developer of the app might want to implement a certain policy that limits the number of accesses to the resource to a finite value, to prevent any third-party script included in the app from abusing this resource.

HybridGuard provides the option of limiting the number of resource accesses per principal for any specified resource. As shown in Listing 3.6, using the `bound_check(principal, resource, action)` API provided by HybridGuard, a developer can ensure that access is disallowed if the bound limit is reached. The bound-limit is specified as part of the policy specification.

Allow List policy. In the same app mentioned above, a developer might want to restrict the app to send text messages to a certain list of numbers (resource-based allow list). In another scenario, a developer might simply want to restrict a principal to access only a certain list of resources (principal-based allow list). As Shown in Listing 3.6, HybridGuard provides the developer with the `allowlist_check(principal, resource, action, args)` API, using which the developer can ensure that these specific restrictions are met. The allow list is specified as part of the policy specification.

History-based policy. Some policies cannot be expressed as just static permissions or access control rules. To prevent exfiltration of sensitive user data, the developer might want to enforce a policy that disallows access to potential data-exfiltration channels, such as `SMS`, `Email`, or any form of network access—if an untrusted principal has accessed a sensitive device resource, such as `Geolocation`. HybridGuard allows the developer to define and track principal-based local security states to enable the runtime enforcement of such stateful and fine-grained policies. Fig. 3.4 depicts a simple history-based policy.

Custom policy. Since our framework is written in JavaScript, the developer can express numerous custom policies that can not be specified using the above mentioned policy classes. For example, a developer might want to enforce a policy where any third-party JS is not allowed to create an invisible `iframe`.

3.6 Experimental Results

In this section, we present the results of our experimental evaluation. The core code of HybridGuard is a JS program enclosed inside an anonymous function `(function(){ /* code */})();` to protect the code and its security states. The monitor and policy engine are combined within this anonymous function comprising of ~800 lines of JavaScript code. To deploy HybridGuard in a hybrid mobile app, the developer needs to copy this library together with the JSON polcy specification file to the `www` folder of the app, then include it in the main HTML page (`<script src="HybridGuard.js"></script>`) right after the core JavaScript library of the hybrid app (`cordova.js` in the case of Cordova app). As discussed earlier, to include a JS

file (local or remote) the developer can use our API `loadJSwithPrincipal(principal,url);` to load and execute code under a principal, instead of including these files using the conventional `<script>` tag. This loading code can be implemented in a separated JS file after `<script src="HybridGuard.js"></script>`) or can be placed at the end of `"HybridGuard.js"` file outside the anonymous function. After assigning principals for different JS files, the developer can edit the JSON policy specification file to define fine-grained permission for each principal. To evaluate the effectiveness of HybridGuard, we have tested it with a self-developed hybrid mobile app and a few real-world Android apps from Google Play.

3.6.1 Testing on self-developed hybrid mobile app

We use Cordova framework (version 5.3.3) to develop the testing app. We include several resource plugins listed in Table 3.1, such as SMS, email, contacts, camera, geolocation, accelerometer, File System and develop their functionality in local JavaScript files, and load them with "local" principal using `loadJSwithPrincipal("local",<js-file>);`. We also host similar JavaScript files remotely and load them with "remote" principal using `loadJSwithPrincipal("remote",<remote-js>);`. We specify the principal-based permission in the JSON file to allow/disallow access to the resource by a principal. We have performed several minor modifications in the policy code to make it consistent with the plugins and policies. All policies introduced in the previous section have been implemented. We use Cordova to build the app for both Android and iOS platforms. For Android, we deploy the app directly to real devices Nexus 5X and Nexus 6P running on the Android 7.1.1 (Nougat). For iOS, we use Xcode (version 7.2.1) to build and deploy the app to an iPhone 6s Plus iOS 9.2 simulator. We use debug messages to observe if the principal propagation is tracked correctly. The permissions to the device resources are checked at runtime correctly based on principal. Fine-grained policies such as information flow and history based policies are soundly enforced. We note that Cordova has been used for our testing, however, since HybridGuard is developed in JavaScript, it can be easily adapted and applied to other hybrid mobile frameworks with some trivial modifications in the enforcement and policy code.

Table 3.1: List of Policies Enforced on Plugins

Resource	PlugIn and Resource object	Method	Policy Enforced
Files	cordova-plugin-file Object: window.requestFileSystem	requestFileSystem	Allow List History-based Policy
Camera	cordova-plugin-camera Object: navigator.camera\end{tabular}	getPicture	No Send after read
Contacts	cordova-plugin-contacts Object: navigator.contacts	find	Allow List History-based Policy Resource bounds policy
Accelerometer	cordova-plugin-device-motion Object: navigator.accelerometer	getCurrentAcceleration watchAcceleration	Allow List Enforcement
SMS	cordova-sms-plugin Object: sms	send	Allow List History-based Policy Resource bounds policy
Geo Location	cordova-plugin-geolocation Object: navigator.geolocation	getCurrentPosition watchPosition	History-based Policy Resource bounds policy
Video Recording	cordova-plugin-media-capture Object: navigator.device.capture	captureVideo captureImage	Allow List
Secure Storage	cordova-plugin-secure-storage Object: cordova.plugins.SecureStorage	SecureStorage SecureStorage.get SecureStorage.set	History-based Policy Allow List

3.6.2 Testing on real-world Android hybrid apps

We have performed a small-scale evaluation on real-world Android hybrid apps by manually downloading a few Android apps from `apkpure.com`, that are hybrid and also available on Google Play. We use `apktool` (https://github.com/iBotPeaches/Apktool), a reverse engineering tool for APKs, to decode resources to nearly original form (use e.g., `apktool decode -f -s apkFile.apk`). We include the framework library, i.e., `HybridGuard.js` and the policy specificaiton (JSON file) to the `www` folder, and modify the main page to include the library and load the core scripts. Similarly, in the testing app, we do some minor modification in policy code to adapt the APIs. After this modification to the `www` folder, we rebuild the app using the `apktool` (use e.g., `apktool build modifiedApkFolder/`). The app is then signed using `jarsigner` (`jarsigner -verbose -keystore your.keystore modifiedApkFile.apk`) and is installed on the device.

We have downloaded ten hybrid mobile app APKs and modified them by manually including HybridGuard as described above. A few apps that have been tested successfully are Parked Car Locator, Web Ratio, Remote SMS Control, Graded, Fan React, My Car Navigator.

These applications access various system resources like Camera, Geo Location, Accelerometer, Contacts, or File System. Policies like limiting the access to resources or send messages and location details only to allow listed sources, blocking SMS and email sending as soon as a content from a file is read have been enforced. The tested apps with enforceable security policies are listed in Table 3.2.

Table 3.2: List of tested hybrid mobile apps

Application Name	**Resources Accessed**	**Policies**
Parked Car Locator	Geo Location	Allow List Enforcement
My Car Navigator	Geo Location Accelerometer	Allow List Enforcement Resource bounds policy
Fan React	Contacts SMS	Allow List Enforcement History-based Policy Resource Bounds Policy
Graded	SMS Contacts File System	Allow List Enforcement History-based Policy Resource Bounds Policy
Remote SMS Control	SMS Contacts File System	Resource bounds Policy Allow List Enforcement History-based Policy
Web Ratio	Contacts File System	Allow List History-based Policy Resource Bounds Policy

3.6.3 Performance

Tests to identify app performance and overhead after injecting HybridGuard into the app have not been performed yet. However, while manually testing the app, we did not notice any significant performance issues or delays in app processing. Prior work on similar JS interception reports that the overhead of these implementations is not significant [19, 20, 25, 24, 26].

3.7 Conclusion

We present the design and implementation of HybridGuard, a robust framework to specify and enforce principal-based fine-grained security policies to guard against attacks in hybrid mobile apps originating from third-party JavaScript. Our enforcement framework is platform independent as it is developed in JavaScript; thus it can be deployed on various mobile platforms and hybrid development frameworks without modifying them. We have

demonstrated the implementation of the policy engine and specification of the principal-based and fine-grained policies. We specify a wide range security policies that the app developer can use to mitigate potential attacks. We have conducted experiments to evaluate the framework and policies on real hybrid apps and mobile devices.

Our in-scope threats come from potential malicious third-party JavaScript code in a hybrid app that a developer explicitly includes; therefore, our framework relies on developers on defining security policies. In practice, the app users might be in a right position to define desired security policies to protect themselves. In future work, we also plan to extend the policy system so that the app users can specify their policies on a hybrid app. We also plan to construct a testbed of hybrid apps and an ontology of possible attacks so that we can conduct a large-scale evaluation of real-world hybrid apps and effective security policies.

CHAPTER 4: EXTENSION—HYBRIDGUARD: A MULTI-PARTY, FINE-GRAINED PERMISSION AND POLICY ENFORCEMENT FRAMEWORK FOR HYBRID MOBILE APPLICATIONS[1]

4.1 Introduction

In the previous chapter we introduce a novel policy enforcement framework for hybrid mobile apps. The proposed framework can enforce principal-based, stateful policies, on multiple origins without modifying the hybrid frameworks or mobile platforms. We demonstrate how our policy enforcement framework can detect and prevent potential malicious behavior to protect the security and privacy of users. In this chapter, we discuss new efforts towards improving HybridGuard's design and implementation and conducting a more comprehensive set of experiments [56].

4.1.1 Changes to Design and Implementation

We revise the policy specification to allow the storage of runtime parameters consistently in a file, rather than in memory as in the previous design. This allows changes, such as revoke a granted permission or user customization of the parameters after app installation. We update the enforcement code accordingly so that it reflects the new design. We discuss updates to the fine-grained permissions and policy model under §4.2. We discuss updates to the implementation according to the new design in §4.3.

We also introduce a list of new policy templates based on the new design of the policy language. These policy templates are novel contributions to the literature and we expect these new templates will have significant impact on the research community and also on the industry. These templates are not only applicable for hybrid mobile apps, but they can also

[1]This chapter includes previously published ([56]) joint work with Phu Phung, Rakesh Reddy, Steven Cap, Anthony Pierce, and Meera Sridhar

be adapted and deployed to in-lined reference monitor implementations in similar domains , such as web or cyber-physical systems. We further elaborate this in §4.4.

4.1.2 New Experiments

We perform significant new experiments and report new results in the following parts:

1. **Compatibility.** We implement a base test app and build it using different development frameworks for two major platforms—Android and iOS. We test the revised design and implementation and report the results, which demonstrate that HybridGuard can be easily integrated with these frameworks and platforms to enforce policies.

2. **Real-world apps evaluation.** We evaluate the usability of the framework by integrating it with already available real-world Android apps by reverse-engineering the code and injecting HybridGuard to enforce policies. We evaluate HybridGuard on 40 real-world Android hybrid apps downloaded from several app stores and perform our tests on a real Android device. Combined with the evaluation from the previous chapters, HybridGuard has been evaluated with 50 real-world Android hybrid mobile apps that demonstrate the successful integration of our framework.

3. **Performance.** We execute the app variants and measured the time difference posed by our framework, between original apps and policy enforced apps. We report these performance results as a new contribution to this work.

The experiments and results are discussed in §4.5.

We have made significant extensions to our previous work [55] with the following new contributions:

- We extend the specification of multi-party permissions and policies to support user-centric usage control to protect users' privacy. We present practical permission and policy patterns that developers can deploy in hybrid mobile apps to prevent potential real-world attacks and privacy violations.

- We implement a proof-of-concept prototype that stores the pre-defined policy templates permanently in local storage. This approach ensures that policy states can be updated persistently. It also supports the customization of policies, i.e., end-users can personalize the policy parameters at the installation or runtime.

- We perform significant evaluations and report practical experimental results on various aspects. Our framework is platform-agnostic, since it is compatible with various hybrid app development frameworks and two major mobile platforms (Android and iOS). We show that practical policies can soundly prevent attack scenarios while posing lightweight overhead. We demonstrate that our framework is also applicable to real-world hybrid mobile apps.

Roadmap. The rest of the chapter is organized as follows. Section 4.2 discusses our policy specification design and updates to the specification design from our preliminary work [55]. Section 4.3 provides details about the updated policy manager. Section 4.4 discusses the new policy classes that HybridGuard can enforce, and §4.5 provides details about the new evaluation.

4.2 Updated specification of multi-party, fine-grained permissions and policies

In this subsection, we describe the policy specification design and illustrate how to apply these policies in realistic scenarios. Our goal is to specify rules on how JavaScript code from different parties interact with device resources and users' sensitive information. To this end, our policy specification supports two types of policies as described below.

4.2.1 Multi-party and context-aware permissions

We extend the permission model in mobile architecture. Our new permission scheme allows developers to define and enforce context-aware permissions for each party on a single granted permission. For each resource access or action, i.e., granted permission, developers can define which party can access/perform action on that resource under a label "principal". We support not only `allowed` or `denied` for each principal per resource, but also provide access qualifiers

such as *read, write*, and *create*. We also support context-aware properties such as *allow list and bound* in this permission specification. Our specification ensures that a granted permission must be monitored at runtime so that it will not compromise the security of the app and the privacy of the user by any party. Our novel permission model overcomes the limitations of "all-or-nothing" conventional permission in mobile that open the possibilities for attacks as discussed in §3.2.1.

We use JavaScript Object Notation (JSON) to specify our multi-party and context-aware permissions. We express each device resource in an array element inside a JSON file, each of which has an array of permission objects, identified by a principal (the label for a party). For each resource, developers can specify which principal can be allowed with further runtime constraints. For instance, Listing 4.1 illustrates an example of multi-party permission that allows the local code (loaded with principal "`trusted.com`") to read and write on the contact resource with several restrictions, while allowing JavaScript code from "`untrusted.com`" read-only permission. JavaScript code loaded with other principals is denied access to this resource by default in this example. This specification is an extended version of our preliminary work [55], where more restrictions are defined. In particular, as shown in the example in Listing 4.1, a granted permission is restricted to runtime constraints such as the number of access times, duration, block list, or allow list. We elaborate these new constraints as templates in §4.4. The motivation of this specification is to allow users to change the principal restrictions to enable a more customized fine-grained policy tailored to them.

```
{"resources": [{
  "name": "contacts",
  "permissions": [{
    "principal": "trust.com",
    "read": "true",
    "write": "true",
    "maxUseLimit" : "..",
    "currentUseLimit": "..",
    "maxTimeLimit" : "..",
    "currentTimeLimit": "..",
    "longitude": "..",
    "latitude": "..",
    "distanceAround": "..",
```

```
14      "blacklist" : ["..","..",..]
15     },
16     {
17      "principal": "untrusted.com",
18      "read": "true",
19      "write": "false"
20     }]
21  //...
22  }
```

Listing 4.1: An abbreviated example of fine-grained permissions and policies for two origins

4.2.2 Updated stateful and Fine-grained Security Policies

Multi-party and context-aware permission can enforce policies that control code from a source to access a granted resource. However, permission-based policies cannot capture and prevent potential malicious actions such as sensitive information leakage or UI attacks, as we discussed in the motivating attack examples. In addition to the multi-party permission check, our framework also allows developers to define custom and fine-grained policies such as allow list specification, stateful, and history-based policies. In this framework, we use JavaScript code to define these custom policies. For example, to prevent potential information leakage, developers can define a policy that "after a principal reads the contact list (assume that the principal is allowed to read the contact list in principal-based permission), it is not allowed to send any SMS". Listing 4.5 illustrates this type of policy. We note that this policy is also based on multi-party: the principal violating the aforementioned example policy is denied to send SMS, but other principals such as the first-party code can still be allowed to send SMS.

Privacy-based and custom policies. Since our fine-grained policy specification can capture potential malicious actions at runtime, our framework can be used to protect the privacy of users. Since HybridGuard is developed in JavaScript, developers can express any custom policies that cannot be generalized in rules. In §4.4, we provide a wide-range of policy templates in the structure of multi-party permissions presented previously. Depending on a specific app and its third-party code, developers can use all or select parts of the template to deploy in the hybrid app at the development stage so that the policy can be enforced and

customized by users at runtime.

4.3 Updated Policy Management and Enforcement

In our previous prototype implementation [55], we store the specification in a local JSON file within the app, and load it using `XMLHttpRequest` into a JSON object to perform principal-based permission checks. In this extended specification (cf., §4.2), we need to keep and update runtime parameters, e.g., the number of accesses, therefore, this JSON object needs to be updated and synchronized consistently. To this end, we revise the previous implementation by loading the policy template, provided at the development stage, and store it within a data directory of the app for the first time. Storing in a data directory allows the file can be updated as all files within an app at the installation time are read-only. The first step is to check if the file is already present in the data directory. If it does not exist, we load the original JSON template file and store it in the new location. Otherwise, we use that existing file for policy checking and updating. Pseudo-code (for brevity as the real code is in an asynchronous version with more processing steps) in Listing 4.2 illustrates this process.

```
1 function loadPolicy() {
2     var dirEntry = getDataDirectory();
3     var policy = dirEntry.getFile("policy.json");
4     if(!policy){
5         policy=loadPolicyTemplate();
6     }
7     return parsePolicy(policy);
8 }
```

Listing 4.2: Pseudo-code (for brevity) to load internal policy specification

Let us consider a scenario when a user installs a hybrid app that integrates our framework. As a norm, the user needs to grant permissions requested by the app. With our framework, ideally, the user can define more fine-grained restrictions or customized policies such as "revoke a granted permission for an origin in the app". Also, the user should be able to customize some policy parameters to protect her own privacy. Our current implementation allows users to customize policies by editing the file content directly at runtime as we store the policy specification in a data directory. However, understanding and defining policies

in JSON specification is not an easy task, especially for layman users. In the future, we plan to map this policy specification into user interfaces so that end-users can easily edit the parameters at the installation phase or runtime.

4.3.1 Updated Policy Manager

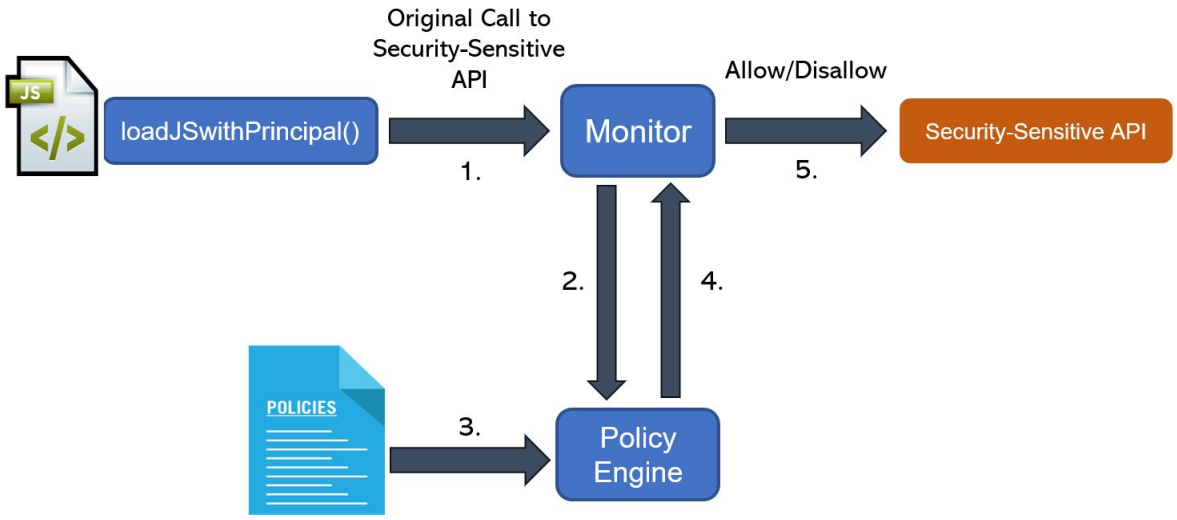

Figure 4.1: HybridGuard's components and policy enforcement

HybridGuard's monitor intercepts API calls accessing a resource, as depicted in Fig. 4.1. Therefore, for each API invocation, the monitor invokes the Policy Manager to check the policies to allow or disallow that API. The monitor code maps an API call to an action defined in the policy specification so that the Policy Manager can perform the check to return the decision.

There are two layers of checking for this Policy Manager module. An API call is allowed and executed if it passes both of these two layers of checking. The first layer is to check the multi-party and context-aware permission in the JSON object cached in memory. We synchronize this cached object with the policy specification file stored in the data directory to ensure that all policy states are updated persistently. Our framework also performs the same synchronization mechanism when the end-user customizes existing policies on the fly. For example, when the user disallows a resource for an origin by editing the policy file content, the new content is loaded into the JSON cached object. With this synchronization, we

ensure that the runtime monitor enforces the newly updated policy when the app invokes a corresponding API call.

For each policy pattern (cf. §4.4) defined in the JSON specification, we implement a corresponding function to look-up the permission based on the resource and principal (the caller) and check the policy parameters based on the context. The Policy Manager module also updates runtime parameters, such as the number of times when an API call is allowed and executed.

The second layer of checking is the custom policies defined purely in JavaScript. The implementation of these checks is dependent on each policy category. In the next section, we introduce the policy patterns and templates, together with its implementation details that support this Policy Manager module.

4.4 Updated Policy Patterns and Templates

Implemented as a reference monitor, our framework supports fine-grained security policies satisfying safety property of execution, i.e., preventing bad things from happening as it is implemented as a reference monitor. These fine-grained policies can be leveraged to protect the `privacy` of users. In this section, we present a wide range of policy templates that developers can use to deploy in hybrid apps at the development stage, depending on the functionality of the app. These policy categories cannot be expressed in current coarse-grain permission models, and are novel compared to our previous work [55]. Table 4.1 elaborates how these policy templates can be deployed and enforced for common device resources. These devices resources include bridge APIs, i.e., plugins provided by a hybrid development framework, and native objects shared by developers as discussed in §3.3. An API comprises a resource object and a method in the corresponding columns in Table 4.1. For example, the API to send SMS messages comprises the object "`sms`" (from the plugin "`cordova-sms-plugin`" provided by Cordova-based frameworks) and the method "`send`". As illustrated in the table, this API, i.e., `sms.send`, can be enforced with four different policy categories, including volume bound, duration usage, history-based, and location-based, as described

Table 4.1: List of Policies Enforced on Plugins

Resource	**PlugIn and Resource object**	**Method**	**Policy Can Be Enforced**
Files	cordova-plugin-file Object: window.requestFileSystem	requestFileSystem	Volume-bound Duration usage Allow List/blacklist History-based
Camera	cordova-plugin-camera Object: navigator.camera	getPicture	Volume bound Location-based History-based
Contacts	cordova-plugin-contacts Object: navigator.contacts	find	Volume bound Allow List/Blacklist History-based
Accelerometer	cordova-plugin-device-motion Object: navigator.accelerometer	getCurrentAcceleration watchAcceleration	Duration usage Location-based Volume-bound
SMS	cordova-sms-plugin Object: sms	send	Allow List/blacklist History-based Location-based
Geolocation	cordova-plugin-geolocation Object: navigator.geolocation	getCurrentPosition watchPosition	Volume-bound Duration usage History-based Location-based
Video Recording	cordova-plugin-media-capture Object: navigator.device.capture	captureVideo captureImage	Duration usage History-based Location-based
Secure Storage	cordova-plugin-secure-storage Object: cordova.plugins.SecureStorage	get set	Volume-bound Duration usage Allow List/blacklist History-based

in the following subsections. Listing 4.3 shows an example of volume bound policy for this `sms.send` API on two principals. we illustrate an example of the `sms.send` API interception in Listing 3.2, where the `PolicyCheck` function is a part of the Policy Manager module to handle these policy templates, as described in §4.3.1.

4.4.1 Configurable context-aware permission-based policies

In this subsection, we introduce fine-grained policies based on permissions, which developers can deploy at the development stage; however, end-users can personalize this at the installation stage or runtime.

4.4.1.1 Volume bound policy

Many mobile apps abuse device resources by frequently invoking the device resources, such as reading the contact list a hundred times, as demonstrated in the litureture [80]. In some

scenarios, a user might want to limit the volume of resource usage, such as the number of SMS messages an app can send per day.

Our specification supports such a volume bound policy within a time unit. In our current specification, we support "day" as the time unit; however, it can be extended to support other time units, such as an hour, or a week. We define this policy using the "maxUseLimit" property of the policy specification. To enforce this policy category for a device resource over a principal, developers need to keep and set the value for the field `"maxUseLimit" : ".."` in the JSON specification, together with the field `"currentUseLimit": ""` with an empty value as illustrated in Listing 4.1 (Line 7–8). Developers can use this pattern to define policies for any device resource as listed in Table 4.1. For example, developers may want to enforce a fine-grained restriction over a granted "`geolocation`" permission that allows local code to read it at most 5 times per day, and limits the code from "`google.com`" to read at most once per day. Such a policy can be specified for two different principals ("`local`", and "`google.com`") over a single resource "`geolocation`", as illustrated in Listing 4.3. We note that, by default in our enforcement mechanism, any code without principal information will be disallowed access to resources even if its the user granted the permission.

```
1  {"resources": [{
2    "name": "sms",
3    "permissions": [{
4     "principal": "local",
5     "read": "true",
6     "maxUseLimit" : "5",
7     "currentUseLimit": ""
8    },
9    {"principal": "google.com",
10    "read": "true",
11    "maxUseLimit" : "1",
12    "currentUseLimit": ""
13   }]
14 //...
15 }
```

Listing 4.3: An example of volume bound policy

4.4.1.2 Duration usage policy

Mobile users might want to limit the duration that a device resource, such as accelerometer, geolocation, or video recording, can be used to save energy or to protect user's privacy. Our policy specification supports this policy category with "`maxTimeLimit`" property for each principal. Similar to the previous category, our current prototype implementation supports the duration per day and the time unit in minute, although these are extensible. Similarly, developers need to keep and set the value for the field `"maxTimeLimit" : ".."` in the JSON specification, together with the field `"currentTimeLimit": ""` with an empty value. The policy specification illustrated in Listing 4.4 allows the local code to access the geolocation for 10 minutes and limits the code from "`google.com`" to 1 minute.

```
{"resources": [{
   "name": "geolocation",
   "permissions": [{
    "principal": "local",
    "watch": "true",
    "maxTimeLimit" : "10",
    "currentTimeLimit": ""
   },
   {"principal": "google.com",
    "watch": "true",
    "maxTimeLimit" : "1",
    "currentTimeLimit": ""
   }]
//...
}
```

Listing 4.4: An example of duration usage policy

4.4.1.3 Location-based policy

Some policies might be related to location, i.e., allowing a device resource access at particular places. For example, users might want to allow sending SMS messages only while the device is in domestic. We support this policy category with a coordinate ("`latitude`" and "`longtitude`" property) and a distance ("`distanceAround`" property) as illustrated in Listing 4.1.

4.4.1.4 Block List/Allow List Policy

In some scenarios, a principal is allowed to invoke an API with parameters. For example, to send an SMS message, the code needs to call `sms.send` with the number to be sent together with other parameters. Users might want to allow (allow list) or disallow (block list) a principal to send SMS to a limited list of receivers. To support this, we provide `"allowlist"` and `"blocklist"` properties in the specification as shown in Listing 4.1 that developers can deploy and end-users can customize the list.

4.4.2 Custom Fine-grained Policies

Implemented in JavaScript, HybridGuard can enforce fine-grained policies expressed in JavaScript code that can be defined by developers at the development phase. We present history-based policy and generic web-based policy templates that can prevent potential attacks.

4.4.2.1 History-based Policies

A common attack by malicious JavaScript is to read sensitive user data and send it to the attacker through different channels, such as the `src` attribute of the `<img>` HTML tag. Although CSP can prevent some of these channels so that the leakage can be limited, there are other channels specific to a mobile device that are not captured by CSP, such as SMS, and email. Developers can prevent this potential information leakage by monitoring the access to sensitive information and preventing access to certain APIs that are not captured by CSP. For example, developers can define a policy "no SMS sending after contact list is read" by intercepting the contact read action and toggle the contact read flag, which can be checked in the policy for SMS send–if the flag is toggled, the SMS send action is disallowed. This whole policy is defined in Listing 4.5.

4.4.3 Web-based Security Policies

There are several other potentially malicious behaviors of third-party JavaScript code, such as manipulating the DOM and create UI attacks, such as touchjacking (e.g., by creating an invisible `iframe`) or launch a phishing attack. Using HybridGuard, in addition to the

```
var contact_read_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"contacts", "read"))
        return; //no permission for this principal
    toggle(contact_read);// update the contact read history
    if(!bound_check(p, "contact", "read") return;
    return proceed();//allow the invocation
  };
var sms_send_policy = function(args, proceed) {
    var p = getTopofPrincipalStack();
    if(!principal_permission_check(p,"sms", "send"))
        return; //no permission for this principal
    if(contact_read) return;
    if(!bound_check(p, "sms", "send") return;
    if (!allowlist_check(p, "sms", "send", args[1])) return;
    return proceed();//allow the invocation
  };
intercept(sms, 'send', sms_send_policy);
intercept(navigator.contacts, 'find', contact_read_policy);
```

Listing 4.5: Example of "no SMS send after reading contact list"

supported policies presented above, the app developer can implement any custom policies in JavaScript when intercepting HTML5/DOM APIs and JavaScript bridge APIs. In the touchjacking example, the developer can enforce a policy that disables the creation of an invisible `iframe`.

4.5 Evaluation

In this section, we report the evaluation of the proposed framework including the experiments and results on the functionality, compatibility on different hybrid app frameworks, mobile platforms, and real-world hybrid apps, the performance and overhead, and its security. We release our prototype and experimental results on https://github.com/sridhar-research-lab/hybridguard-2019.

4.5.1 Compatibility

We evaluate the compatibility of our framework in two settings: a test suite and existing hybrid apps on an app store. First, we develop a test suite of variants of a hybrid app in multiple hybrid development platforms with standard bridge APIs to access device resources. We deploy our framework on these app variants to evaluate how our framework works in these

settings. In the second evaluation setting, we want to test how our framework is compatible with existing hybrid apps in the wild. To this end, we use Android real-world hybrid apps since we can reverse engineer Android apps to inject code and rebuild the apps. We describe the experiments and their results below.

4.5.1.1 Test suite

We first develop a base hybrid app using four different hybrid app development frameworks, including Cordova v6.2.3, Framework7 v1.6.4, Onsen UI v2.4.2, and Intel XDK v3987. To test the functionality, we include corresponding plugins, including SMS, email, contacts, camera, geolocation, accelerometer, and file system in each framework. We list these resources and their corresponding APIs in the first and second column of Table 4.1. This inclusion is to ensure that the base app can use common device resources. We write JavaScript code in a `.js` file and include it locally into the app to use the plugins to access the device resources. We also host the `.js` file remotely and include the remote script into the app as a third-party code. We use each framework to build a variant of the app for both Android and iOS platforms and deploy them to real devices.

Before integrating our framework to the app variants, we build and deploy them to physical devices to ensure that the apps are functional on these devices. For Android, we deploy the app variants directly to a Google Pixel XL device with Android 7.1. For iOS, we use Xcode 9 to build and deploy the app variants to an iPhone 7 Plus device with iOS 10.0.1. We test these eight variants of the app on the two devices. As the Onsen UI variant does not work on the iOS device, we deploy and test them on an iOS 10.0.1 emulator. In all of these testing environments, the app functionally works as expected, and all of the device resources can be accessed properly for both local and remote scripts.

To evaluate the compatibility of our framework, we modify each original app variant to deploy the framework. We first customize the policy template for each app variant and store it in a JSON file within each app folder together with the framework library `.js` file. We specify the multi-party permission in the JSON file to allow/disallow some access to the

resource by a principal. For simplification but still, in general, we define two parties with two principal labels for this permission. We have performed several minor modifications in the policy code to make it consistent with the plugins and policies. We have implemented all policies introduced in the previous section. We then revise the main HTML code of the variants to include the framework library, and replace the existing script inclusion, *i.e.,* `<script src=".."></script>` by the `loadJSwithPrincipal` function provided in our framework to load the JavaScript code with a principal, for both of the local and remote scripts. For example, the original code loading of `<script src="http://remote.com/code.js"></script>` will be replaced by:

`<script>loadJSwithPrincipal("remote.com","http://remote.com/code.js"); </script>`.

For this compatibility evaluation, we define policies to monitor and log the execution. This evaluation is to test if the apps integrated with our framework work as in their original versions. We then rebuild the app variants and deploy to the devices again to test the functionality. We turn on debug messages so that we can observe all the execution logs from our framework. The logs demonstrate that our enforcement code intercepts and monitors all the calls to the device resources. Also, the principals of the code (based on the source) are identified correctly for both local and remote scripts. The functionality of the app variants is preserved. Among the app variants, we note that there is a minor issue in Framework7 on both Android and iOS devices, that the principals are not tracked in the same order. However, access to the resources are functional and monitored by the policies. Fig 4.2 illustrates this compatibility evaluation. As we can see from the figure, our framework is compatible with every framework on the two major mobile platforms, Android, and iOS.

4.5.1.2 Real-world Android hybrid apps

By design, developers need to integrate our framework at the development stage to define and enforce policies. However, to evaluate the compatibility and usability of our framework, we integrate our framework into existing real-world hybrid apps. As Android apps allow us to reverse-engineer the code, we select the Android platform to test our framework. We

Cordova		Framework7		OnsenUI		Intel XDK	
Android	iOS	Android	iOS	Android	iOS Emulator	Android	iOS
✓	✓	✓	✓	✓	✓	✓	✓

Figure 4.2: Compatibility crossing frameworks and platforms of the modified app with HybridGuard embedded

first collect real-world Android-based hybrid mobile apps by downloading these apps in `.apk` files from a third-party app store (https://apkpure.com/) using a scripting program. We filter the apps to select hybrid mobile apps for our evaluation. We use the `apktool` tool (https://github.com/iBotPeaches/Apktool) to reverse-engineer the hybrid app APK files. This step helps in obtaining the entire web code and resources of the hybrid apps. We write a simple scripting program to identify apps that access device resources such as camera, geolocation, accelerometer, contacts, filesystem, or storage through included JavaScript files. Before integrating our framework, we rebuild these apps back to APK files to install and run on an Android device. To do this, we use the `apktool` tool to rebuild and then self-sign the apps with our own generated keys (we use the `jarsigner` tool to do this). Using these preparation steps, we select 40 Android hybrid mobile apps that both include JavaScript files to access device resources and function correctly after the repackaging process without modifying the app code.

Next, we integrate our framework into these apps, following a similar step as done for the test suite described above. In particular, we copy the framework library (the `HybridGuard.js` file) and permission JSON file to the `www` folder within each app's folder. We use the same general JSON permission for every app and define several fine-grained policies for testing. The classes of policies implemented include resource-bounds (e.g., *Access to SMS resource only five times a day*), history-based (e.g., *No network access after accessing geolocation*) and white-list policies (e.g., *Only specific principals can write to contacts*). We include the framework script into the main HTML file (usually the `index.html` file) and modify the script

inclusions using our loading interface. We rebuild these modified apps again and install them on the same Android device to test. We successfully test on the 40 Android hybrid mobile apps, demonstrating that the apps modified with our framework preserve the developer's intended functionality. Also, our execution logs show that our framework suppresses the calls to security-sensitive APIs that violate any policy. These results evidence that our framework is not only compatible with real-world hybrid apps, but also soundly enforce the defined policies for these apps. We publish this dataset, including the original APK files, the modified app folders with our framework, and the modified APK files on https://github.com/sridhar-research-lab/hybridguard-2019/tree/master/evaluation/realAndroidapps.

4.5.2 Fine-grained policy enforcement

In the second round of evaluation, we revise the policies for the app variants in our test suite to evaluate whether our framework can soundly enforce these policies. Our test policies do not only log the execution but also to monitor the behaviors of the execution with fine-grained policies as provided in the templates presented in the previous section. These policy templates include multi-party and context-aware permissions in the JSON specification that can prevent the attack scenarios of abusing device resources, as identified in §3.2.1. We also define custom fine-grained policies in JavaScript. These custom policies are to prevent potential attacks such as malvertisements and sensitive information leakage as well as UI redress attacks, as also discussed in §3.2.1. To test the effectiveness of our policy enforcement framework, we modify the script code to intentionally violate the policies at some points and rebuild and deploy the apps. Experiments and logs confirm that the accesses to resources are functional until the policies are violated, demonstrating our framework enforces the defined policies correctly. For example, we enforce a volume bound policy that allows the maximum of 5 times of SMS sending as illustrated in Listing 4.3. Our test code repeatedly calls the SMS sending API in every ten seconds to send an SMS message. The first five messages were successfully sent from the app and received on another phone. After that, our framework stops the execution of this SMS API and alerts a message, as shown on the left of Fig. 4.3.

The other test cases in Fig. 4.3 illustrate the correct enforcement of other policy categories, including duration usage, and location-based, respectively.

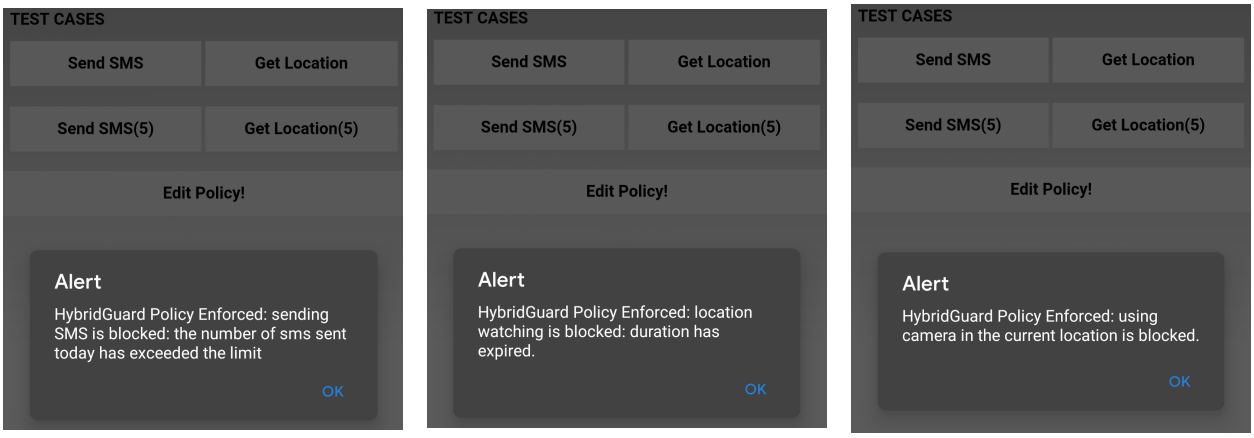

Figure 4.3: Policy enforcement evaluation on different policy categories

4.5.3 Performance

We evaluate our framework performance by measuring the runtime overhead posed by our policy enforcement mechanism. Typically, the runtime overhead of web-based systems like hybrid mobile apps can be measured by both in JavaScript operations, i.e., micro-benchmarks and the load or render time, macro-benchmarks [157, 26]. We measure the load time of an app with and without our framework. We do not notice any slowdown as the load time of the original app and the modified app with our framework are almost identical. This result can be explained by the fact that JavaScript code in e.g., hybrid apps is mostly event-based, and asynchronous[2]. For this reason, we are interested in evaluating the micro-benchmarks of operations that do not depend on triggered events, including getting the current position, acceleration, and direction. To this end, we modify the code in original app variants to execute these operations 1000 runs, to achieve high precision, and measure the time before and after the runs. For each case, we run the apps on the two devices with ten trials to get the averaged numbers.

We then integrate our framework in these apps with three different policies, including usage

[2]See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

limit, the number of times per day, and the duration of execution time (cf., §4.4). We set very high limits in these policies to ensure that no violation will happen. Thus the operations are just executed as usual. We do the same measurements as in the original apps to get the corresponding averaged numbers. We report the overhead by showing the slowdown ratio over 1000 runs between the average execution time of each operation test with the HybridGuard-integrated app and that of the original app for each combination of a development framework and mobile platform. Table 4.2 shows these slowdown ratio numbers for each operation on the combination of three development frameworks (Cordova, Framework7, OnsenUI) and two mobile platforms (Android and iOS). Although our framework is compatible with Intel XDK as demonstrated in §4.5.1.1, the time measurement over 1000 runs on the app based on this framework, both the original and modified app, was inconsistent in 10 trials. Therefore, we exclude the Intel XDK framework in performance evaluation results.

Overall, our experimental results evidenced that our HybridGuard framework only poses a small additional runtime overhead on 1000 runs, as shown in Table 4.2. However, there are no common patterns for the overhead of each operation crossing various frameworks and devices posed by our framework. In particular, for the acceleration operation, our framework has almost no overhead, crossing the three tested hybrid development frameworks and two mobile platforms. For the get current position operation, we see that the overhead of our framework for this operation is quite small for Android. At the same time, they vary in iOS for different hybrid development frameworks. For the get direction operation, our framework poses nearly no overhead for the app variants in iOS with Cordova and OnseiUI frameworks, a small overhead for all apps in Android, but surprisingly high overhead for Framework7 app in iOS. Interestingly, we have observed that each app execution time in each Android and iOS device is a significant difference. For example, our framework overhead on the acceleration operation is almost the same for Android and iOS, crossing hybrid framework, as shown in Table 4.2. However, the execution times in each platform are vastly different, as visualized in Fig. 4.4.

Table 4.2: The slowdown ratio over 1000 runs of typical device resource operations. Numbers in each cell represent the slowdown ratio of an operation on a development framework (including Cordova, Framework7, OnsenUI) and mobile platform (including Android and iOS).

Resources/APIs	Cordova		Framework7		OnsenUI	
	Android	iOS	Android	iOS	Android	iOS
Current Position	2.03	2.89	1.37	2.01	1.44	4.22
Acceleration	1.04	1.07	1.16	1.00	1.13	1.03
Get Direction	4.59	1.14	1.09	5.97	1.85	1.08

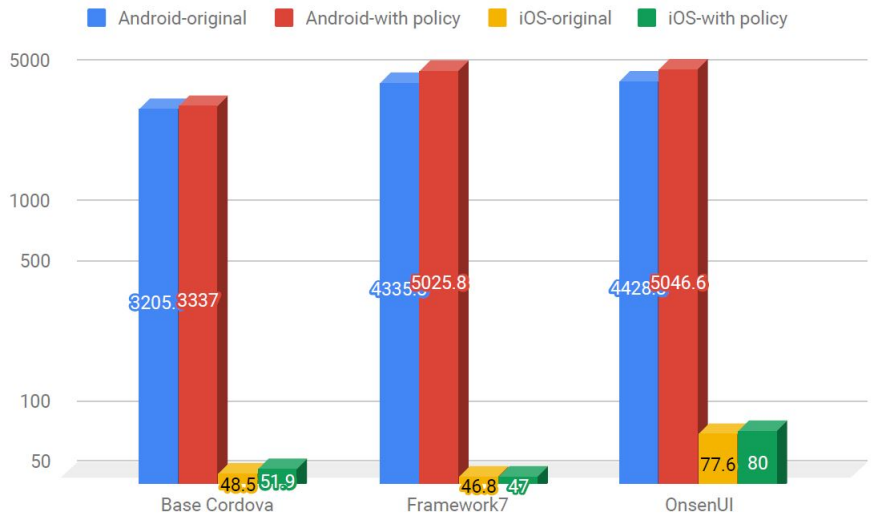

Figure 4.4: Overhead of the acceleration operation posed by our framework crossing development frameworks and two mobile platforms.

4.5.4 Security Analysis

As discussed earlier, potential code injections and information leakage attacks by the web channels can be eliminated by the standard Content Security Policy (CSP) in hybrid mobile apps. Our framework provides an extra layer of protection on JavaScript code that is allowed by CSP. As required by default CSP, developers have to define each JavaScript code in a `.js` file, either for first-party or third-party code. HybridGuard provides a new JavaScript API to obtain the content of these `.js` files and execute them under a principal. This approach requires the code to run before other first-party or third-party code in the app so that our library has the highest priority to control the behaviors of the loaded code. This mechanism ensures that our enforcement code is tamper-proof. As described in the

implementation section, we protect the enforcement code and security states of our framework within an anonymous function, which is inaccessible from outside code. Access to JSON policy specification files is prohibited from unauthorized principals, enforced by the monitor. Therefore, the integrity of our framework is guaranteed. We ensure the *complete mediation* of JavaScript web APIs by systematically exploring and mediating all their possible aliases and channels generating JavaScript code on the fly. This protection is a known technique from prior work [156]. For JavaScript bridge APIs provided by hybrid frameworks, we have to manually identify the possible channels for each API to ensure it is completely wrapped. As we can control the behaviors of the loaded code, any unauthorized access can be detected and prevented.

CHAPTER 5: HYBRIDIAGNOSTICS: AN AUTOMATED VULNERABILITY ASSESSMENT FRAMEWORK FOR HYBRID SMART HOME COMPANION APPS[1]

5.1 Introduction

An essential part of the IoT SmartHome ecosystem is the *IoT companion mobile app*, or briefly, *companion app*, which allows a user to control the IoT device while at home or remotely [158]. For example, fitness companion apps interact with fitness bands to monitor a user's heart rate and other vitals, hotels are piloting smart doors that replace key cards with companion apps, companion apps control automated garage doors, and air conditioning or alarm systems in a smart home can be controlled through a companion app. In addition to completely controlling the IoT device, in most cases, the companion app can also perform high-privilege functions, such as updating the device firmware or changing security codes on specific IoT devices, such as smartlocks. Multiple past incidents and works show that device and companion app security currently does not exert a significant influence on product or infrastructure design [159, 160, 161, 162]. What exacerbates the problem is that most IoT vendors are not software companies and therefore lack comprehensive training in cybersecurity or even foundational software engineering best-practices [162]. Even if vendors choose to outsource companion app development, their options are limited to either freelancers or small to mid-size companies with no dedicated security team [163, 164].

Companion apps are readily available to anyone through major app stores such as Google Play, Apple App Store, and other third-party app stores. The ubiquity of these apps makes them a lucrative target for cyber attackers that are readily trying to identify new vectors for exploiting IoT devices and smartphone users. Past works prove that most companion apps contain at least one potential vulnerability that can be exploited to launch serious cyberattacks against the user or the device [161, 165, 47, 46, 48].

[1]This chapter includes previously submitted ([57]) joint work with Meera Sridhar

In this work, we present HybriDiagnostics, a vulnerability assessment framework that identifies preexisting security issues in companion apps developed for Android using *hybrid* [2] mobile app development frameworks. *Hybrid* mobile apps are currently a popular technology that uses HTML, JavaScript, and CSS for their core business logic and UI and executes in the embedded web browser. A few popular *hybrid app development frameworks* include Apache Cordova [66], React Native [6], Ionic [7], Framework7 [8], Flutter [9], Phonegap [166], Onsen UI [10], and NativeScript [11]. The hybrid app development technology allows mobile apps to be *write-once-run-anywhere*, saving substantial time and resources required to develop separate apps for different mobile platforms.

We use HybriDiagnostics to survey 102 real-world Android-based *hybrid* companion apps to identify preexisting security issues, including exposure to web attacks, misconfiguration of security measures during app development, usage of outdated SDKs, and unsafe usage of DOM elements. Each security issue contributes to expanding pathways to exploit the IoT device and user's privacy. We explore how these companion app security issues affect the IoT ecosystem and demonstrate the consequences through either real-world case studies or synthetic but plausible scenarios. For this work, we focus on apps developed using Apache Cordova, Ionic, Monaca, OnsenUI, Phonegap, and Framework7. These frameworks are widely used by the developer community and share the same software stack and architecture [66, 167, 14, 168]. For simplicity, we refer to apps developed using any of these six frameworks as *Cordova-based* apps in the rest of the paper.

Our main contributions in this work include—

- we present HybriDiagnostics, a vulnerability assessment framework that can identify preexisting security issues in hybrid companion apps;
- we discuss eleven security issues in hybrid companion apps developed using Apache Cordova, Ionic, Monaca, OnsenUI, Phonegap, and Framework7;

[2]The terms *hybrid* and *web-based* are used interchangeably to refer to apps developed using cross-platform tools such as Apache Cordova. We refer to such apps as *hybrid* apps in the rest of the paper.

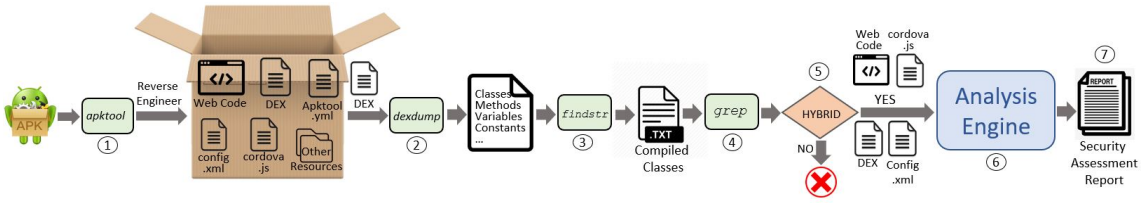

Figure 5.1: HybriDiagnostics Overview

- we analyze 102 real-world hybrid companion apps to identify the presence of preexisting security issues. For each security issue, we present an analysis of the issue conducted on our companion app dataset, results, and either a *proof-of-concept attack* (PoC) or a synthetic attack scenario. The PoC or the synthetic attack scenario demonstrates how these security issues can be exploited in a smarthome environment; and

- we discuss select mitigation techniques and tools to avoid the presented security issues and reduce the attack surface of hybrid companion apps.

Roadmap. The rest of this chapter is organized as follows. Section 5.2 provides an overview of HybriDiagnostics, its Analysis Engine, our dataset for analysis and experiments, and a brief background on how we construct PoC attacks and synthetic attack scenarios. Section 5.3 discusses each of the eleven security issues in detail and presents the results of our analysis for each security issue accompanied by a relevant PoC or a synthetic attack scenario. Section 5.4 discusses select mitigation tools and techniques that developers can use to avoid each security issue while developing a hybrid companion app, and §5.5 concludes.

5.2 Overview

In this section, we provide an overview of HybriDiagnostics. We describe the Analysis Engine, the heart of the HybriDiagnostics toolchain, and we provide information about the dataset used for the analysis and discuss the types of attack scenarios presented in this work.

5.2.1 HybriDiagnostics Toolchain

Fig. 5.1 provides an overview of HybriDiagnostics. Identifying a given app as a hybrid is a major initial task that HybriDiagnostics requires to complete before beginning the hybrid companion app analysis. To identify whether a given app is hybrid, we extend the

work done by Ali et al. to characterize hybrid apps according to the used app development framework [169]. In that work, the authors inspect the `classes.dex` file and search for references to specific Java classes associated with the respective frameworks. However, this work has two limitations. First, the work is limited to PhoneGap, Appcelerator Titanium, and Adobe Air. Second, the approach cannot identify whether an app is hybrid in the case of an obfuscated APK. We overcome both limitations in our hybrid app identification process.

In Step (1), HybriDiagnostics uses `apktool`[150] to reverse-engineer a given APK and obtain the packaged resources, including the `classes.dex`, a Dalvik executable that references classes and methods used within an app. In Step (2), HybriDiagnostics uses `dexdump` [170] to disassemble the DEX file into human-readable format. In Step (3), HybriDiagnostics uses `findstr`, a Windows OS utility, to search the `dexdump` output for `<Class-Descriptor>` strings, and extract fully-qualified class names of all compiled classes. HybriDiagnostics stores the extracted class names in a text file for further analysis. In Step (4), HybriDiagnostics searches the generated text file using simple pattern matching (using `grep`) for references to specific Java classes associated with the respective app development frameworks. HybriDiagnostics overcomes the first limitation of Ali et al.'s work by including all Cordova-based frameworks (Apache Cordova, Ionic, Monaca, OnsenUI, Phonegap, and Framework7) and React Native in the identification process. A few classes that HybriDiagnostics uses to identify different frameworks include `org.apache.cordova` for Apache Cordova, `com.facebook.react` for Facebook React Native, and `org.framework.ionic` for Ionic. HybriDiagnostics acquires the knowledge of which unique classes to search for in a given app to identify the app development framework by reverse-engineering and analyzing demo apps available on each hybrid app development framework's websites. In the case of an obfuscated APK, HybriDiagnostics overcomes the second limitation of Ali et al.'s work by scanning the decompiled APK directory and searching for `cordova.js`, which indicates that the app is Cordova-based since the file is present in all Cordova-based hybrid APKs. HybriDiagnostics does further UI analysis on the HTML code of these apps to identify the usage of framework-specific scripts and tags. For example,

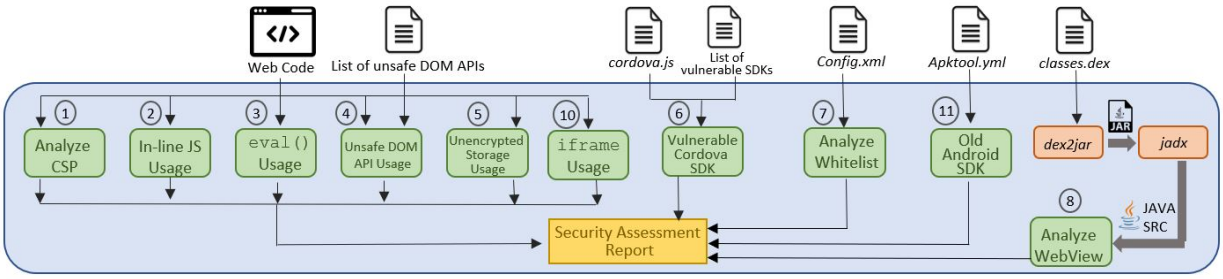

Figure 5.2: HybriDiagnostics Analysis Engine

Ionic apps use framework-specific elements, such as `<IonButton>` in HTML code. In Step (5), if HybriDiagnostics identifies the app as native, then HybriDiagnostics takes no further action; if HybriDiagnostics identifies the app as a hybrid, it sends the app to the Analysis Engine (Fig. 5.2). In Step (6), the Analysis Engine analyzes the, is the heart of the toolchain, and assesses the hybrid companion app APK for preexisting security issues, and generates a vulnerability assessment highlighting the security issues (Step (7)).

5.2.2 Analysis Engine

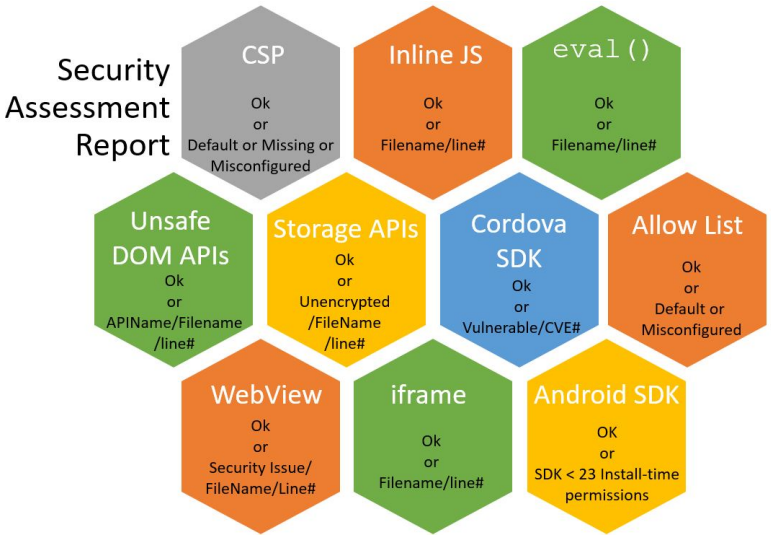

Figure 5.3: Vulnerability Assessment Report

As shown in Fig. 5.2, the Analyses Engine comprises several components that perform the security analysis presented in §5.3. Each component is labeled with a number corresponding to the security issue number reported in §5.3. Components (1), (2), (3), (5), (10) receive the app's web code as input and analyze the CSP, inline JavaScript usage, `eval()` usage, and

unencrypted storage usage, respectively. Component (4) receives the app's web code and a list of unsafe DOM APIs as input and identifies the usage of unsafe DOM APIs in the web code. Component (6) receives the `cordova.js` and a list of vulnerable Cordova SDKs as input and identifies if an app uses a vulnerable Cordova SDK. Component (7) receives the `config.xml` as input and analyzes the allow list usage. Component (11) receives the `Apktool.yml` as input and extracts the Android target SDK. Component (8) receives `classes.dex` file as input and converts it into a JAR file using the tool `dex2jar`. Then, component (8) user another tool `jadx` to extract the Java source code from the JAR file. Then, component (8) extracts the `WebView` configurations from the Java source code and analyzes them. Once HybriDiagnostics finishes analyzing the given app (APK), it generates a report in a format seen in Fig. 5.3.

5.2.3 Dataset

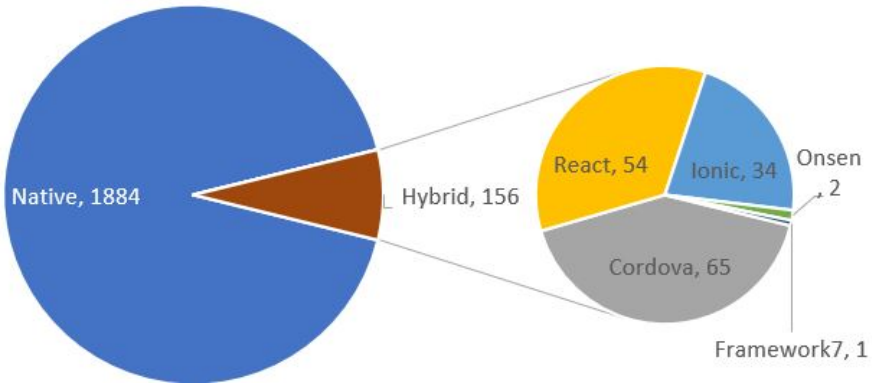

Figure 5.4: App Categorization (2082 apps)

Our dataset consists of 2082 real-world Android-based companion apps (hybrid and native), shared with us by Wang et al. and downloaded from Google Play Store before April 2019. From that dataset, HybriDiagnostics identifies 102 Cordova-based companion apps. HybriDiagnostics uses fresh copies of Cordova-based real-world companion apps, downloaded in April 2021, for the security analysis presented in Section 5.3. The identified apps include 65 Apache Cordova apps, 34 Ionic apps, two Onsen UI apps, and one Framework7 app. HybriDiagnostics does not encounter the usage of Phonegap. For the analysis presented in §5.3.8, *Attacks on `WebView`*, HybriDiagnostics considers the entire dataset (2082 apps),

since a large percentage of the apps in the dataset use `WebView`.

Additionally, HybriDiagnostics identifies 54 React Native apps. However, for the security analysis in Section 5.3, HybriDiagnostics excludes React Native apps since these apps have a different software stack and architecture from conventional hybrid apps [171]—React Native apps use JSX [172] for writing business logic, a syntax extension to JavaScript, over HTML, JavaScript, and CSS. React Native uses native Android APIs to render the UI instead of rendering the UI in `WebView`. JSX is immune to traditional injection attacks that are a major attack vector for conventional hybrid apps [173].

5.2.4 PoC Attacks & Synthetic Attack Scenarios

In our work, we combine PoC attacks and synthetic attack scenarios to demonstrate the exploitability of the security issues in hybrid companion apps in a smarthome ecosystem. We conduct the discussed PoC exploits in a controlled setting—we install the apps on a Google Pixel test smartphone running Android 10 OS and storing no actual user data; we store the remote web page used for the PoC exploit discussed in §5.3.6 on a server that is under our control. We choose the PowerBrick Alarm app by Micron Security [174] for our PoC exploits since HybriDiagnostics identifies this app for the most number of security issues amongst all the apps in our dataset. For §5.3.6 (Vulnerable Cordova SDKs), we choose the Smart Home Security app for Nedis devices [175], since despite numerous other security issues, the PowerBrick Alarm app uses a Cordova SDK with no reported vulnerability. The synthetic attack scenarios that we discuss feature different real-world apps and can eventuate in a smarthome ecosystem if an attacker exploits the identified security issues.

5.3 Analysis of Security Issues in Hybrid Companion App Dataset

In this section, we present an analysis of preexisting security issues in 102 Cordova-based smarthome companion apps. Each subsection explains the security issue, discusses the HybriDiagnostics analysis and results, and presents either a PoC attack or a synthetic attack scenario demonstrating the impact of the security issue in a smarthome ecosystem.

We do not analyze standard JavaScript libraries included in the app, and only focus on

security issues introduced in first-party (developer-written) app code. For identifying standard JavaScript libraries to exclude from the analysis, we follow the approach in Niakanlahiji et al.'s work [176], which uses *Context Triggered Piecewise Hashes (CTPH)*. CTPH, also known as *fuzzy hashes* is a standard in identifying similar files since its algorithm matches sequences of identical bytes in the same order [177]. Even though the bytes between these sequences can differ in both content and length, CTPH is still efficient in identifying similar files. We compute the CTPH value for various versions of the top 200 libraries listed by the `cdnjs` website. In our analysis, before analyzing any JavaScript file, we compute its CTPH value and compare it with the previously generated list of CTPH values to ensure the file under examination is not part of any well-known library.

5.3.1 Default, missing, or misconfigured CSP

Security Issue #1: Cordova provides a default CSP with every project; however, the default CSP allows the usage of inline JavaScript and the `eval()` function, which are well-known to render the app vulnerable to injection attacks [178, 179]. In the case of a default CSP, any injected JavaScript code would execute in the app's context and would have the same privileges as the app [56], exposing the IoT device and the smartphone to dangerous attacks [161].

HybriDiagnostics parses the web code in a given APK to extract the CSP configuration and categorizes the app's CSP into three categories—

- *default*—the CSP allows inline JavaScript and `eval()` and does not specify the location/domain of dynamic resources;
- *missing*—the app does not include a CSP; this often happens because impractical deadlines, business requirements, and shortage of resources prevent developers from implementing an efficient CSP, and they resort to simply deleting the CSP [180];
- *misconfigured*—while it is possible to have a deep level of control over the policy, errors in the definition of directives may lead to unexpected consequences. Prior research

works ([181, 182]) and blog posts ([183, 184, 185]) detail various CSP misconfigurations and its bypasses. Some misconfigurations include using the wildcard for directives, or partial configurations such as defining a `script-src` but leaving out `style-src` or `object-src`. An attacker can use a Flash object to inject JavaScript code into the app and bypass the CSP[186]. Misconfiguration can render the CSP ineffective and vulnerable to injection attacks.

HybriDiagnostics parses all the HTML files of a given app to extract the CSP configurations. Then, it uses the tool *CSP Evaluator* [187], a Google Open Source project, to evaluate if the CSP is misconfigured. CSP Evaluator, a tool based on a large-scale study conducted by Weichselbaum et al. [181], determines if given CSP serves as a strong mitigation against injection attacks such as cross-site scripting. It can identify subtle bypasses that undermine the CSP configuration. We note that since HybriDiagnostics uses CSP Evaluator to evaluate if a CSP is misconfigured, the analysis results depend on the completeness of the misconfigurations that CSP Evaluator can identify.

Results: Out of the 102 Cordova-based apps, only 32 apps implement a CSP, implying that developers of 70 apps chose to delete the CSP. Out of the 32 apps that implement a CSP, ten have a default CSP, and 22 apps have a misconfigured CSP.

```
1 <meta http-equiv="Content-Security-Policy" content="default-src 'self' data: gap: https://
      ssl.gstatic.com 'unsafe-eval'; style-src 'self' 'unsafe-inline'; media-src *">
```

Figure 5.5: Default CSP in Cordova apps

Proof-of-Concept Attack: For this PoC, we use the PowerBrick Alarm app by Micron Security Innovation [188]. The app facilitates local and remote control of multiple alarm systems—fire, medical, and home, and allows IP camera control.

HybriDiagnostics identifies that the app does not implement a CSP. We black-box test the app for XSS by brute-forcing all UI input fields using payloads from the OWASP XSS Filter Evasion Cheat Sheet [189]. We discover the app is vulnerable to XSS via the alarm `name`

field. We inject JavaScript code into the app via the `name` field; since the injected code has the same privileges as the app, we can access sensitive device resources, including camera, location, contacts, and internet. We can successfully track a user's geolocation and send it to a server under our control. We are also able to disable the set alarms.

5.3.2 Inline JavaScript

Security Issue #2: Developers can include JavaScript in an HTML page via two methods:

(i) *External* JavaScript—store the JavaScript code in an external file and include it in the HTML page using the `src` attribute of the `<script>` tag;

(ii) *Inline* JavaScript—embed the JavaScript code directly into the HTML page using the `<script>` tag but no `src` attribute; using a `javascript:` URL, or inline event handlers, such as `onmouseclick` and `onfocus`.

Cybersecurity experts recommend against using inline JavaScript since it requires using the '`unsafe-inline`' CSP directive, which exposes the app to injection attacks [178]. However, previous works that survey traditional web apps show that a substantial majority of CSP-enabled web apps resort to adding the '`unsafe-inline`' directive ([181, 190]). This practice occurs since moving inline JavaScript to external files requires re-structuring the entire app, which is a non-trivial task and downgrades the app's performance due to the synchronous loading of numerous external scripts [191].

HybriDiagnostics parses the HTML files of a given app and identifies the usage of inline script elements by searching for `<script>` tags without a `src` attribute, `javascript:` URLs, and event handlers.

Results: Out of 102 Cordova-based apps in our dataset, 71 apps use inline JavaScript. Out of the 71 apps, only 24 apps implement a CSP; however, all 24 apps use the '`unsafe-inline`' directive.

Synthetic Attack Scenario: For this attack scenario, we consider the i4Home [192] app from our dataset. The app controls and monitors the i4Home Wireless Security Alarm System.

The app implements a CSP but allows the execution of inline scripts. Allowing execution of inline JavaScript can render the app vulnerable to XSS, which can be exploited to take complete control of the IoT device [161]. The i4Home app has a feature to name different rooms in a smarthome via the `Room Name` field. Let us assume there is no input validation on this field. If an attacker injects malicious JavaScript code into the app via the field, it executes instantly since the app allows the execution of inline JavaScript. Since the code executes in the app's context, it will have the same privileges as the app itself. The malicious code can access and exfiltrate sensitive user data and can disable the smarthome alarm system.

5.3.3 Unsafe `eval()`

Security Issue #3: The `eval()` function takes a `String` input and executes it as JavaScript code [179]. Using `eval()` is dangerous and not recommended since it exposes hybrid companion apps to injection attacks [179]. The `eval()` function also requires using the '`unsafe-eval`' CSP directive since a CSP without this directive prohibits the use of `eval()`. Adding the '`unsafe-eval`' directive results in a less efficient CSP, as explained in subsection 5.3.1. Additionally, `eval()` executes in global scope and has access to the entire app code [193].

For this analysis, HybriDiagnostics only considers developer-written code in the app and excludes other standard JavaScript frameworks/libraries included in the app. HybriDiagnostics parses the HTML and JavaScript files of the app code and identifies apps that use `eval()` to evaluate expressions.

We note that `eval()` usage is strictly a vulnerability only if data flows to the `eval()` call from an untrusted input. We leave automating this data flow analysis to future work. For the synthetic attack scenario presented, we manually analyze the app code to identify that the app does not validate untrusted input. Then, we further analyze the app to identify that `eval()` accepts untrusted input that can be modified by the attacker (Fig. 5.6).

Results: Out of the 102 Cordova-based apps in our dataset, 50 apps use `eval()` in the app code.

```
function show_remote_controller_page_receiver(itemid) {

    // clear battery low and tamper icon
    var varname2 = "alarm_" + itemid + "_battery";
    eval("status_model." + varname2 + "='high'");

    var varname3 = "alarm_" + itemid + "_tamper";
    eval("status_model." + varname3 + "='notamper'");
```

Figure 5.6: `eval()` usage in Smart Home Security app

Synthetic Attack Scenario: For this scenario, we consider the Smart Home Security app used in §5.3.6. HybriDiagnostics identifies that the app uses `eval()` at multiple locations in the app code. Fig. 5.6 shows an instance of `eval()` usage in the app. A manual code analysis reveals that `itemid` in Fig. 5.6 stores a user supplied value. Let us assume the app does not adequately sanitize the user supplied value before storing it in `itemid`. An attacker can inject malicious JavaScript code into `itemid` that disables the home alarm system. Since `eval()` considers anything passed to it as code, it executes the malicious JavaScript code instantly jeopardizing the user's safety.

5.3.4 unsafe DOM APIs

Security Issue #4: To display content in hybrid companion apps, developers use *Document Object Model* (DOM) APIs and attributes, and jQuery APIs including `document.write()`, `innerText`, `innerHTML`, `outerHTML`, and `html()`. APIs that consider the passed parameters as data (`String`) and not code are *safe* APIs, and APIs that consider the passed parameters as code and execute it are *unsafe* APIs [1]. If a developer requires dynamically generating HTML elements on the app page, the developer resorts to using the unsafe APIs. However, while using unsafe APIs, a developer should validate any untrusted input, i.e., input not under the developer's control, before processing it. Untrusted input can originate at the client-side, i.e., user input, and can also originate at the server-side, for instance, non-validated input from a database. Section 5.4.4 provides more details on validating user input.

For the analysis, HybriDiagnostics scans the HTML files of a given app to identify usage of unsafe APIs to display app content.

Table 5.1: APIs and attributes used for displaying data [1]

DOM APIs and Attributes	**Safe or Unsafe**	**JQuery APIs**	**Safe or Unsafe**
document.write()	Unsafe	html()	Unsafe
document.writeln()	Unsafe	append()	Unsafe
innerHTML	Unsafe	prepend()	Unsafe
outerHTML	Unsafe	before()	Unsafe
innerText	Safe	after()	Unsafe
outerText	Safe	replaceAll()	Unsafe
textContent	Safe	replaceWith()	Unsafe
value	Safe	text()	Safe
		val()	Safe

Results: Out of 102 Cordova-based apps in our dataset, 84 apps use unsafe APIs to display the app's content.

Proof-of-Concept Attack: For this attack, we consider the PowerBrick Alarm app again (see §5.3.1). As shown previously, this app is vulnerable to an XSS attack via the alarm `name` field. HybriDiagnostics identifies that the app uses jQuery API `html()` to display the alarm `name` in the app. The `html()` API takes the user-supplied alarm `name` as input and displays it in the app. However, the app does not validate this input and executes any code passed to the `html()` API.

5.3.5 Unencrypted storage

Security Issue #5: It is common for mobile apps to store user and app data on the device's local storage. Developers using Cordova-based frameworks can either use core web APIs, such as `window.localStorage`, for storing data on the device's local storage, or use plugins such as `cordova-sqlite-storage` [194]. However, these commonly used storage APIs and plugins do not encrypt the data before storing it and are hence insecure. According to OWASP's Top 10 mobile risks list, unencrypted storage of critical user and app data such as API keys continues to prevail [195].

Any person with physical access to the device or malicious code injected into the app can trivially access unencrypted data on local storage. A few community-developed plugins are available at the developers' disposal using Cordova-based frameworks to store data on

local storage securely. These plugins include `cordova-plugin-secure-storage` [196], `cordova-sqlcipher-adapter` [197], and `com-intel-security-cordova-plugin` [198] (not supported by Intel anymore, but still available to use). These plugins encrypt the data before storing it and provide data security.

HybriDiagnostics parses the web code of a given app and identifies the usage of both insecure and secure API and plugins for storing data.

Results: Out of the 102 apps in our dataset, 92 apps use insecure APIs and plugins for storing data. Only 10 apps use secure storage APIs. All 10 apps use the `cordova-plugin-secure-storage` plugin to achieve this. A few apps in our dataset that do not encrypt data before storing include Arnido Smart Home, Wemo by Belkin, and Panoramic 360Â° CCTV Bulb Camera.

Synthetic Attack Scenario: For this attack scenario, we consider the app Arnido Smart Home [199] that stores unencrypted data on local storage. This app controls various smarthome devices, such as electrical appliances, smart lights, alarm systems, and smart locks, through a single app. Let us assume the app stores a sensitive API token in local storage and uses it for authentication while sending any command to the smart lock. Since the app implements a misconfigured CSP and allows inline scripts, an XSS attack can be used to steal this API token. If an attacker acquires this token, they can change the lock code on the smart lock and physically access the smart home when the user is not around.

5.3.6 Vulnerable Cordova SDKs

Security Issue #6: Since Cordova's initial release in 2009, it has undergone numerous performance and security-enhancing changes. Despite the updates being critical to the app's security, numerous apps still use old and vulnerable SDKs and expose themselves to cyberattacks. An example vulnerability affecting Cordova SDK before version 3.7.2 and 4.x before 4.0.2 allow remote attackers to send data to arbitrary applications via an Android *Intent* [200], bypass the allow list, and connect to arbitrary servers using JavaScript to open network socket connections through Webview. This vulnerability allows an attacker to change

the vulnerable app's start page via a crafted `intent`: type URL [201]. Another vulnerability affecting `cordova-plugin-inappbrowser` allows an attacker to execute arbitrary JavaScript in the main application's Webview [202].

For this analysis, we build a comprehensive list of reported Cordova SDK vulnerabilities (shown in Table 5.2, using the MITRE CVE (Common Vulnerability Enumeration) database and security advisories by the Cordova team [203, 204]. The first column of the table is the *CVE*—a unique ID assigned to each publicly disclosed computer security flaw [205]; the second column is the *CWE* (Common Weakness Enumeration) and it's associated ID—a community-developed list of software and hardware weakness types. It serves as a baseline for weakness identification, mitigation, and prevention efforts [206]; the third column is the *Common Vulnerability Scoring System* (CVSS) [207] score for the vulnerability—a numerical score provided by the organization Forum of Incident Response and Security Teams (FIRST) reflecting the severity of the vulnerability [207]; and the fourth column is the affected Cordova version. HybriDiagnostics extracts the Cordova SDK version from `cordova.js` (Cordova library file) from the given app. HybriDiagnostics checks the identified Cordova SDK version against the list of reported vulnerabilities that we built to identify whether the given app uses a vulnerable Cordova SDK.

Results: Out of the 102 Cordova-based apps, 38 apps use vulnerable SDKs. Some of the apps that use older SDKs with reported vulnerabilities include Wemo by Belkin, Smart Home Security, and Daikin Envi Thermostat.

Proof-of-Concept Attack: For this scenario, we consider the app Smart Home Security [208] by `omguardec2`, available on Google Play Store, with over 10,000 installs. App features include surveillance of the home, energy usage monitoring, controlling devices such as power switch, motion detector, door contact, and the security alarm system. Despite being a smarthome security app with a large customer base, the app was last updated in October 2017 and uses Cordova library version 3.6.4, launched in September 2014.

The app is affected by a reported vulnerability in the `CordovaActivty` [209] class, the main

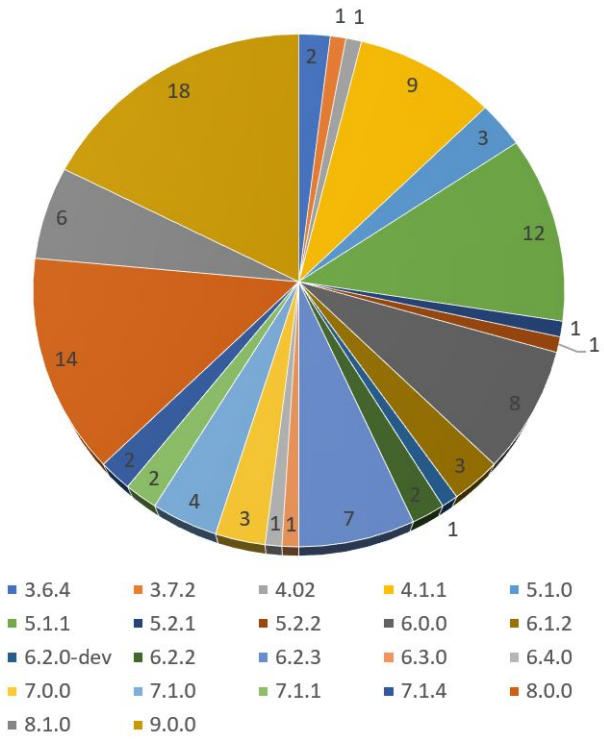

Figure 5.7: Cordova Android SDKs used in IoT companion hybrid mobile apps in our dataset

Android `Activity` [210] of the Cordova application. The vulnerability allows remote attackers to modify undefined Cordova `preferences` (app configuration variables) such as `Fullscreen`, `DisallowOverscroll`, `Orientation`, `InAppBrowserStorageEnabled`, `SplashScreen`, etc., via a crafted `intent:` type URL embedded in an attacker-controlled web page or an app [201, 211].

Successful exploitation of the vulnerability requires two prerequisite conditions to be met: 1. at least one of the app's components (`Class`) extends the `CordovaActivity` class; and 2. at least one of the Cordova-supported `preferences` (except `LogLevel` and `ErrorUrl`) is not defined in `config.xml`. Assuming the above two conditions are satisfied, the app can be exploited by tricking the user into either opening an attacker-controlled web page in the phone browser or installing an attacker-controlled app containing maliciously crafted `intent` URLs. To conduct the exploit, we first manually analyze the app code to ensure that both prerequisite conditions are fulfilled. We then set up a remote web page that contains malicious code to tamper with the app's `preferences` via `intent:` type URLs. Then, we open this web page in Google Chrome on the test smartphone. We can tamper with the app's appearance,

Table 5.2: Apache Cordova Vulnerabilities

CVE ID	CWE ID	Score	Cordova Version
CVE-2017-3160	200	5.8	<6.1.2
CVE-2016-6799	532	5	<=5.2.2
CVE-2015-8320	200	5	<3.7.0
CVE-2015-5256	264	4.3	<4.1.0
CVE-2015-5208	20	4.3	<4.0.0
CVE-2015-5207	284	7.5	<4.0.0
CVE-2015-1835	20	2.6	<3.7.2 and 4.x before 4.0.2
CVE-2014-3502	200	4.3	<3.5.1
CVE-2014-3501	254	4.3	<3.5.1
CVE-2014-3500	17	6.4	<3.5.1
CVE-2014-1884	264	7.5	<= 3.3.0
CVE-2014-1882	264	7.5	<= 3.3.0
CVE-2014-1881	264	7.5	<= 3.3.0
CVE-2012-6637	20	7.5	<= 3.3.0

inject pop-ups and text, inject splash screens, and crash the app, causing denial-of-service. We can achieve this even without the app running.

5.3.7 Default or misconfigured Allow List

Security Issue #7: For this analysis, HybriDiagnostics categorizes a given app into two categories—

- *default*—network requests can be made to any origin (`<access origin="*">`), and the app can use any Android `intent` type URL to ask the system to open the respective system app. *Intent* [200] is a messaging object used to request an action from another app; `intent` type URLs allow this through `WebView`;

- *misconfigured*—Cordova recommends to securely configure all three allow list types to allow access to specific network domains and sub-domains, limit the allowed intents according to the app requirements, and using a CSP over Network Request Allow List since a CSP allows more fine-grained control over the network requests an app can make. Cordova also specifies that the Navigation Allow List takes precedence over the Intent Allow List. We build a list of ambiguities and errors in the allow list definition

to identify allow list misconfigurations. For instance, a wildcard for Navigation Allow List (`<allow-navigation href="*" />`) renders the Intent Allow List ineffective since it implicitly captures all intents. Another instance can be defining the Navigation Allow List as `<allow-navigation href="https://*/*" />`, in which case `WebView` can be navigated to any URL starting with `https`.

A default or misconfigured allow list leaves the app vulnerable to a variety of attacks, such as *phishing* [212], *drive-by-downloads* [213], and *data-exfiltration*. For this analysis, HybriDiagnostics first extracts the allow list configurations from `config.xml` and compares it to the list of misconfigurations to identify a misconfigured allow list. For Network Allow List, if an app defines a CSP, we consider the Network Allow List as ambiguous and report it.

Results: Out of the 102 Cordova-based apps, 43 apps have a default allow list, and 58 apps have a misconfigured allow list. Only one app (Home Alerts—works with Nest) implements a policy with all the three allow lists distinctly defined.

```
<allow-navigation href="https://*.nest.com" />
<allow-navigation href="https://nesteralerts.com" />
<allow-navigation href="https://*.google.com" />
<allow-intent href="https://nesteralerts.com" />
<allow-intent href="https://*.google.com" />
<allow-intent href="http://*.google-analytics.com" />
<access origin="https://*.nest.com" subdomains="true" />
<access origin="https://nesteralerts.com" subdomains="true" />
<access origin="https://*.google.com" subdomains="true" />
```

Figure 5.8: Allow List configuration (partial) of Home Alerts—Works with Nest app

Proof-of-Concept Attack: For this attack, we consider PowerBrick Alarm app used in §5.3.1. The app provides the user with an option to sign in using a Facebook account. The app has a misconfigured allow list policy with wildcards used for all three allow lists, allowing any injected code to redirect the `WebView` to any URL.

As shown in Fig. 5.9, we leverage the previously discovered XSS vulnerability in the app (see §5.3.1) and create a PoC phishing web page that simulates the Facebook login page. In step (1), the attacker injects JavaScript code in the app to redirect the `WebView` to the phishing web page. In step (2), the injected code navigates the `WebView` to the phishing

Figure 5.9: Missing allow list leads to credential compromise

web page. In step ③, once the user enters their credentials and clicks the `Log In` button, the attacker receives the login credentials.

```
<img src=x onerror=``window.location('https://attacker.com')''>
```

Figure 5.10: JS payload to exploit mis-configured allow list using XSS

5.3.8 `Webview` Attacks

Security Issue #8: Even though the integration of `WebView` in companion apps allows developers to build mobile apps using web technologies, it exposes apps to several attacks, demonstrated in both academic works [214, 215, 216], as well as blog posts [217, 218, 219]. According to Checkmarx [220], the top four insecure coding practices while implementing `WebView`, which lead to most attacks, are—

1. Loading arbitrary third-party content—an app that displays third-party content in `WebView` is potentially harmful since `WebView` runs as a single process and any malicious content in `WebView` has the same privileges as the app itself [217, 218].

2. Loading content over `http` (non-encrypted)—as discussed in the presented attack scenario, loading content over `http` exposes the app to a potential *man-in-the-middle* [221] attack.

3. Enabling execution of JavaScript in `WebView`—by default, the OS disables JavaScript execution in `WebView`. The developer can enable JavaScript execution by using the `setJavaScriptEnabled()` function, however, cybersecurity experts recommend preserving the default behavior if the app does not require client-side scripting. Disabling JavaScript execution in JavaScript ensures the app is resilient to code-injection attacks.

4. Enabling access to local file storage—the function `setAllowFileAccess()` can be used to enable `WebView`'s access to local file storage. However, if the app does not properly validate input, enabling access to local file storage can lead to unauthorized file access via a file traversal attack [217].

For this analysis, HybriDiagnostics scans Java source-code files of a given app to identify the discussed insecure coding practices in `WebView`'s implementation. Additionally, we identify several apps that pass `String` parameters instead of the actual URL to the `loadUrl()` function of `WebView` class. As mentioned in §5.3.3, since we leave dataflow analysis for future work, we cannot confirm whether each `String` parameter is a URL. The results do not include such apps. Therefore, the number of apps that load content over `HTTP` or pass JavaScript code to the `loadUrl()` function could be higher than we currently report.

Results: Out of 2082 apps in our dataset, HybriDiagnostics identifies that 1019 apps use `WebView`; 125 apps use `WebView` to load local content (`file://` URL); 565 apps use `loadUrl()` function of the `WebView` class to execute JavaScript; 808 apps have JavaScript enabled in `WebView`; 111 apps load content over `http`; and 232 apps that enable `WebView`'s access to local storage of the device.

Synthetic Attack Scenario: We consider the app My Leviton [222] by Leviton Manufacturing Co., Inc. The app can control electrical appliances connected to smart switches manufactured by the same vendor. The app loads content over `http` in `WebView`, which exposes the app to a potential man-in-the-middle attack (see Fig. 5.11 for the synthetic attack scenario).

Figure 5.11: Exploiting misconfigured `WebView`

For this attack scenario, let us assume the firmware on a connected smart light bulb updates via the companion app, a common practice among IoT devices [223]. In step (1), the app requests a vendor-controlled remote server for the firmware update URL. The attacker intercepts this request (step (2)) and serves a malicious response (step (3)) containing a crafted firmware update URL. Since the app loads content over `http`, unlike `https`, the content is unencrypted (plain-text) and the attacker can easily modify it. In step (4), the phone accesses the malicious firmware update URL, and in step (5) downloads the firmware. In step (6), the firmware is sent to the smart light bulb. Once the malware enters the light bulb, it can do malicious activities such as increasing the voltage and causing the light bulb to explode and injure the user, switching the light bulb on and off without the user's intent, or spreading to other devices on the same network.

5.3.9 Broken Same-origin Policy

Security Issue #9: In hybrid companion apps, since local app code adds Cordova plugins to `WebView` (in Android), they have no web origin in `WebView`'s context. Therefore, any web content, benign or malicious, loaded into the `WebView` from any origin can directly invoke the Cordova plugins added to the app. Hence, any malicious JavaScript loaded into the `WebView` can access device resources via the Cordova plugins and exfiltrate private user data; SOP fails to protect against such attacks.

Figure 5.12: Broken SOP in hybrid mobile apps

Results: HybriDiagnostics identifies all 102 Cordova-based apps in our dataset as affected by this security issue.

Synthetic Attack Scenario: We consider the Smart Home Security app again (see §5.3.6). As in the case of most free apps, this app displays in-app ads to generate revenue through clicks and referrals [142].

For this attack scenario (see Fig. 5.12) , let us assume the app includes an ad syndicator script to display in-app ads. In step (1), the attacker compromises the attack network by exploiting a known vulnerability in the ad server. Then, in step (2) the attacker modifies the ad syndicator script to access the device resources. In step (3), the malicious ad script accesses the user's sensitive information and stealthily exfiltrates security-sensitive user data to external servers using the SMS and email device resources.

5.3.10 `iframe`s

Security Issue #10: An `iframe` is an inline frame or a rectangular region in an HTML document to embed (display inside) a separate document [224]. A common usage of `iframe`s is embedding videos or displaying ads inside an HTML document. The developer can add an `iframe` to a web page either—

(i) *statically*—using the `<iframe>` HTML tag. Here, the `src` attribute of the `<iframe>` tag defines the URL of the document to embed; or

(ii) *dynamically*—using JavaScript's `document.createElement()` method (see Listing 5.1, Line 1), which takes as a parameter the HTML element's name that the developer wants to create dynamically. The developer can specify the URL of the document to embed in the `iframe` using the dot (.) notation (object property accessor) to define the `src` property of the `iframe` object (Line 3).

```
1  var iframe = document.createElement('iframe');
2  var html = '<body>Foo</body>';
3  iframe.src = 'data:text/html;charset=utf-8,' + encodeURI(html);
4  document.body.appendChild(iframe);
```

Listing 5.1: Adding `iframe` to a web page dynamically

In standard web applications, an `iframe` displays content from the specified URL, but SOP prevents any script in the parent page from accessing the framed page's contents and vice-versa. SOP enforces this behavior to protect the parent page's integrity and isolate potentially malicious documents from compromising the user's privacy.

In hybrid apps, to display content in an `iframe` in `WebView`, the developer requires adding the URL of the document to embed in the `iframe`, to the Navigation allow list. Without adding the URL to the allow list, Cordova does not allow the content to load in `WebView` since the `iframe` is now a part of `WebView` and the Navigation allow list applies to all content in `WebView`, including `iframe`s. Content served in the `iframe` from the allow listed URL has the same privileges as the app itself, hence, can access the Cordova plugins [225]. SOP cannot block this access because once the developer adds the required plugins to the app, the plugins become part of the app's local (on-device) code and have no web origin. Therefore, malicious content served in an `iframe` can access security-sensitive resources via plugins added to the app code. For instance, if a developer adds a third-party ad network to the Navigation allow list and serves an ad in an `iframe`, a malicious ad from the ad network can access security-sensitive device resources via the Cordova plugins.

HybriDiagnostics scans the given app's HTML files to identify `iframe` usage by searching for the `<iframe>` tag in the HTML code. Then, HybriDiagnostics extracts the value

of the `src` attribute of the `<iframe>` tag, i.e., URL of the embedded content, for further analysis. For dynamic `iframe`s, HybriDiagnostics scans a given app's JavaScript code, excluding standard JavaScript frameworks and libraries, and searches for instances of `document.createElement('iframe')`. If HybriDiagnostics finds any such instances, it extracts the `iframe`'s URI by further scanning the code and searching for the line of code that sets the value of `object.src` (see Listing 5.1, Line 3). We then manually analyze the extracted URI to determine whether the content served by the `iframe` is under a third-party's control and also identify the third-party.

Results: Out of the 102 Cordova-based apps, six apps use `iframe`s to display content. Manual analysis of these seven apps shows that only one app, Rogers Smart Home Monitoring app, embeds content in an `iframe` that is not under the control of the vendor. The app embeds a video from `vimeo.com`.

Synthetic Attack Scenario: For this scenario, we consider the Rogers Smart Home Monitoring [226] app in our dataset. The app's features include surveillance, energy usage monitoring, controlling devices such as smart lights, smart switches, smart locks, IP cameras, and alarm systems. The app uses an `iframe` tag to embed third-party JavaScript from https://www.tagmanager.google.com. Let us assume an attacker compromises the third-party server and modifies the content served in the `iframe`. The malicious code can then access device resources via Cordova plugins since SOP does not protect plugin access. The attacker can access and exfiltrate sensitive user data. Additionally, if the app stores the lock codes for the smart locks in local storage, the attacker can also access that, compromising user safety.

5.3.11 Outdated/vulnerable Android SDKs

Security Issue #11: Android OS and SDKs receive regular updates to enhance functionality and patch existing vulnerabilities [227]. Both Google and third-party security researchers identify numerous security issues and vulnerabilities in Android components each month, therefore Google releases monthly security patches and recommends (and constantly reminds) app developers to update app SDKs [228]. Even if a smartphone runs the latest Android OS

Table 5.3: Vulnerabilities affecting each Android SDK in our dataset

Android Version	**API**	**# CVEs**	**Avg. CVSS Score**	**Top 2 CWEs**
Android 2.3.3	10	19	6.95	Information Exposure,Permission and Access Control
Android 3.0	11	15	7.34	Permission and Access Control, Information Exposure
Android 3.1	12	14	6.9	Permission and Access Control, Improper input-validation
Android 3.2	13	13	7.1	Permission and Access Control, Information Exposure
Android 4.0	14	46	4.65	Information Exposure, Permission and Access Control
Android 4.0.3	15	231	7.33	Permission and Access Control, Information Exposure
Android 4.1	16	21	9.41	Memory Corruption
Android 4.2	17	229	7.33	Permission and Access Control, Information Exposure
Android 4.3	18	230	7.35	Permission and Access Control, Information Exposure
Android 4.4	19	216	7.25	Permission and Access Control, Information Exposure
Android 5.0	21	304	7.26	Permission and Access Control, Information Exposure
Android 5.1	22	367	7.46	Memory Corruption, Permission and Access Control
Android 6.0	23	627	7.2	Information Exposure, Memory Corruption
Android 7.0	24	667	7.16	Information Exposure, Permission and Access Control
Android 7.1	25	229	7.23	Information Exposure, Memory Corruption
Android 8.0	26	382	6.95	Information Exposure, Out-of-Bounds Write
Android 8.1	27	254	6.94	Out-of-Bounds Write, Out-of-Bounds Read
Android 9.0	28	148	6.73	Out-of-Bounds Write, Out-of-Bounds Read
Android 10.0	29	266	4.82	Out-of-Bounds Read, Out-of-Bounds Write

version, if an app uses outdated SDKs with API-level vulnerabilities, it is still exploitable [227].

However, even the monthly security patches do not guarantee that every Android phone gets the patch as soon as the patch's release. The reason for this is that Google releases security patches for Android Open Source Project (AOSP), the stock Android OS project. However, other Android phone manufacturers such as Samsung, Sony, LG, and Motorola customize the AOSP by adding additional features and functionality and prepare their own custom Android OS. These manufacturers have to integrate the released security patches into their version of Android OS and release them as system updates [229]. Some manufacturers release more frequent and timely security patch updates for their phones; however there can exist long patch lags [229]—if it is not a Google phone (such as the current Pixel), it can be months before phones receive security updates [228].

Additionally, Google recently announced the end of support for Android OS 7.0 (SDK version 24) and lower, which also includes end of security patches for these SDKs [230]. With both Google and the phone manufacturers not releasing security patches for SDKs 24 and lower, soon attackers can exploit zero-day vulnerabilities in these SDKs to launch

cyberattacks against users. At the time of writing this paper, Google also requires all new apps to target SDK 29 and app updates to target SDK version 28 [231]. As mentioned in §5.2.3, since we re-download fresh copies of our hybrid companion app analysis dataset in September 2020, we can conclude that apps that target SDK versions lower than 28 (less than Google's requirement) in our dataset are not updated by the developer. We conduct a two-fold analysis here—

- Part A—we use HybriDiagnostics to identify the Android SDK version distribution in our entire dataset of 2082 apps (native and hybrid). The SDK distribution provides us with a general idea of the amount of effort IoT code producers put in updating the apps.

- Part B—for any given hybrid companion APK, HybriDiagnostics classifies the APK as *vulnerable* in the security assessment report if it targets Android SDK version 24 or lower. HybriDiagnostics does this because SDKs lower than 24 will not receive security patches anymore and are vulnerable to zero-day attacks. HybriDiagnostics also adds a security warning to the vulnerability assessment report of a given APK if a given APK does not meet Google Play's API requirements and targets an SDK version higher than 24 but lower than 28.

To identify the target SDK version, our first approach is to get the required information from `AndroidManifest.xml` [232]. Every Android app project must have an `AndroidManifest.xml` file (with precisely that name) at the root of the project source. The manifest file describes essential information about an app to the Android build tools (part of SDK), the Android OS, and Google Play. A manifest file can contain several elements that include `<application>` [233], `<uses-permission>` [234], `<uses-sdk>` [235], and others. Google Play relies on the `<uses-sdk>` attribute in the app manifest, to filter an app for devices that do not meet its platform version requirements. The `<uses-sdk>` element has three important attributes—`android:minSdkVersion`, `android:targetSdkVersion`, and `an-`

`droid:maxSdkVersion`. However, declaring the `<uses-sdk>` is not mandatory. While parsing the `AndroidManifest.xml` [232] to identify the target SDK version, we discover that numerous apps do not declare this information in the manifest file. We resolve this issue by extracting this information from another file, `Apktool.yml`, generated during the APK reverse engineering process by `apktool` (see Fig. 5.2 (11)).

Results: For Part A, Fig. 5.13 shows the distribution of the different Android SDK versions in our dataset, represented as a pie chart. Only one app out of 2082 apps targets the current SDK version. Additionally, Table 5.3 shows (terms CVE, CVSS, and CWE have been discussed in §2.2)—(i) the various versions of Android SDKs in our dataset of 2082 apps; (ii) number of vulnerabilities affecting each SDK (obtained from MITRE's CVE database [205]); (iii) Average CVSS score. We average the CVSS score of all the vulnerabilities associated with an SDK version; and (iv) top two CWE categories affecting each SDK version.

For Part B, out of the 102 Cordova-based apps, HybriDiagnostics identifies 59 apps that target Android SDK 24 or lower. For the second part of the analysis, HybriDiagnostics identifies 43 apps that target Android SDK versions higher than 24 but lower than 28. A few hybrid companion apps that HybriDiagnostics identifies as vulnerable include Intelligent Home Center, Home Alerts—works with Nest, Blossom Smart Watering, and Daikin ENVi Thermostat.

Synthetic Attack Scenario: For this scenario, we consider the app Blossom—Smart Watering [236] in our dataset. The app controls a smart watering system and targets Android SDK version 19. The app features a *Profile* page where a user can upload a display picture in various formats such as `JPEG` and `BMP`. Since the image is uploaded as a stream of bytes, the Android SDK features an image parsing library that parses the image bytes to re-form the image.

Let us assume an attacker discovers a buffer overflow [237] vulnerability in the component of the image parsing library that parses JPEG images. Since the app uses SDK version 19 now, it will not receive a vulnerability patch. By uploading a specially crafted JPEG file to

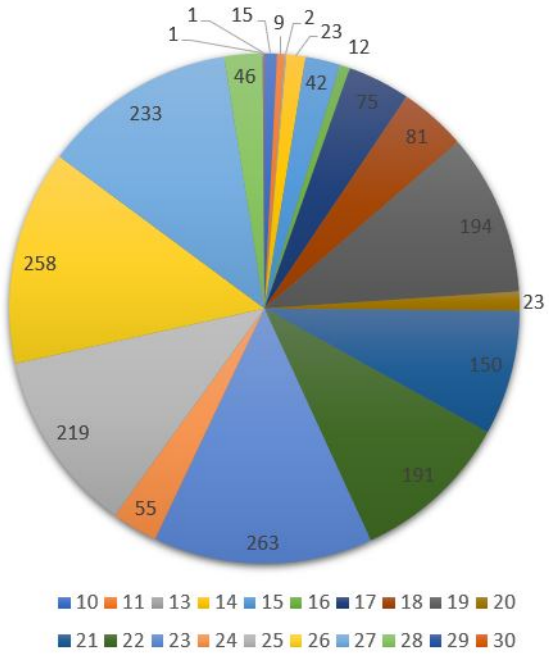

Figure 5.13: Target Android SDKs (1893 apps)

the app, the attacker can trigger the buffer overflow vulnerability and potentially execute arbitrary code on the user's smartphone. The injected malicious code can control the smart watering system, resulting in a lot of water wastage that can be hefty on the user's pocket. More severe consequences include the malicious code can access arbitrary memory locations allocated to other apps running on the device and exfiltrate security-sensitive data, or obtain root access to control the device [238, 239] completely.

5.4 Selected Mitigations

In this section, we discuss tools and techniques that hybrid app developers can use to mitigate the security issues presented in §5.3. Table 5.4 summarizes the mitigation techniques discussed. Each row of the table is divided into four columns—the number assigned to the security issue (see §5.3), security issue (see §5.3), select mitigations, and a reference for additional details on each mitigation. Each subsection briefly describes the security issue then discusses the appropriate mitigation technique that the developer can implement to avoid the security issue.

Table 5.4: Select mitigation for the presented security issues

#	Security Issue	Mitigations	Reference
1	Misconfigured CSP	Chrome Dev Tools & CSP Evaluator	[240, 187]
2	Inline JavaScript	Hash or Nonce	[68]
3	Unsafe eval()	`window.Function`	[241]
4	Unsafe DOM APIs	Input Sanitizers	[242, 243, 244]
5	Unencrypted Storage	Encrypted Storage	[196, 197, 198]
6	Vulnerable Cordova SDK	Updating Cordova SDKs	[245]
7	Misconfigured Allow List	Secure Allow List	[69]
8	Misconfigured WebView	Secure Configuration of WebView	[246]
9	Broken SOP	HybridGuard	[37, 56]
10	iframe	no `iframe`, HybridGuard	[225]
11	Old Android SDK	Update Android SDKs	[227]

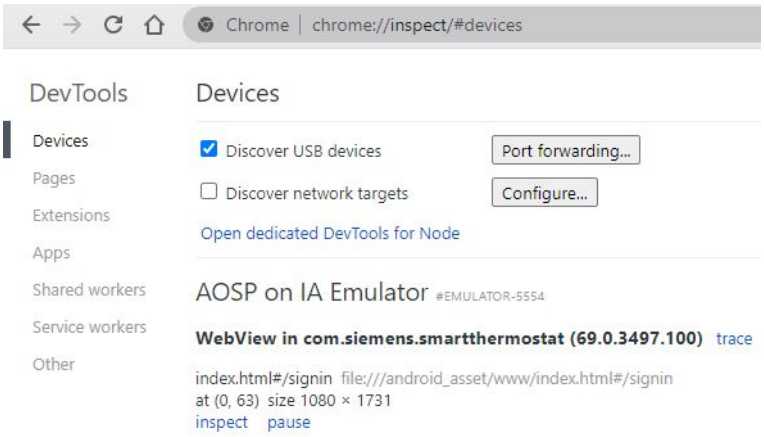

Figure 5.14: Inspecting a hybrid companion app using Chrome Dev Tools

5.4.1 Chrome Dev Tools and CSP Evaluator (Security Issue #1)

As discussed in §5.3.1, in hybrid apps, the default or a misconfigured CSP exposes an app to injection attacks. To prevent this security issue, a developer can do the following—

- *Chrome Dev Tools*—the Google Chrome browser provides a simple but effective way to identify CSP violations in hybrid apps [240]. To debug an app using Chrome Dev Tools, the developer can either use the Android Studio emulator or a real device (Android OS) to run the app. In the case of a real device, the device requires a connection to the machine running Google Chrome via USB. Once the app is running, as seen in Fig. 5.14, the developer can open the URL `chrome://inspect/#devices` in Chrome to access the app debug page. The developer can debug the app and see CSP violation by

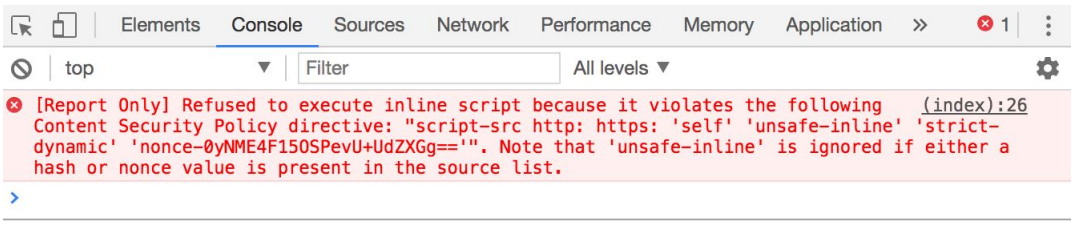

Figure 5.15: Identifying CSP violation using Chrome Dev Tools

clicking the **inspect** button, which opens a new window with the app screen on one side and Chrome Dev Tools on the other. The developer does require to browses the app and try different app features to trigger the CSP violations. Fig. 5.15 shows an example CSP violation in the Chrome Dev Tools Console tab.

- CSP Evaluator—as previously mentioned in §5.3.1, CSP Evaluator [187] allows developers to check if a CSP serves as a strong mitigation against injection attacks. It assists developers in identifying subtle CSP bypasses [186, 183] which render the policy ineffective, discussed in §5.3.1.

5.4.2 Hash and Nonce (Security Issue #2)

```
1    <script>alert("Hello World!");</script>
```

Listing 5.2: A simple script

As discussed in §5.3.2, ideally, developers should avoid using inline scripts since it requires adding the '**unsafe-inline**' directive to the CSP. Besides allowing execution of inline JavaScript, the '**unsafe-inline**' directive allows any injected script to execute in the browser, exposing the app to injection attacks. However, moving inline JavaScript to external files in an already existing app is not a trivial task and breaks the app structure [191]. CSP provides two features to safely include inline JavaScript in an app, which the developer can use [68]—

- *Hash*—for each inline JavaScript code block used in the app, compute its hash value and add it to the CSP. Adding the hash value of the inline JavaScript code block to the CSP restricts the execution of inline JavaScript to only those specific code blocks, disallowing execution of any injected JavaScript, hence, preventing injection attacks [68].

Listing 5.2 shows a simple script that the developer can hash and add to the CSP (Listing 5.3).

```
1  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' '
     sha256-vtOwtCfiL2B+TrRWnLTdfTIr7KTaqohZywH93jHLSGw='">
2
```

Listing 5.3: Adding script hash to CSP

- *Nonce*—in the case where the inline code block contains dynamically generated data, the hash value of the block can change. For instance, in Listing 5.5 Line 3, value of `id` is dynamic and the code block receives the value at runtime. Each distinct value of `id` results in a different hash value of the entire code block. As an alternative, the developer can instead use a *nonce*, a dynamically generated random string independent of the content of the inline JavaScript code block. The developer will need to add the nonce to both the CSP (Listing 5.4) and the inline code block (Listing 5.5) [247]. Before the browser executes any inline JavaScript, it will compare the nonce of that specific inline code block with the nonce in the CSP, and only allow the execution to proceed if both nonce values are equal.

```
1  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' 'nonce-
     VVJJcG9ydHMuY29tIGlzIHRoZSBiZXN0';">
```

Listing 5.4: Adding a nonce to CSP

```
1  <script nonce="VVJJcG9ydHMuY29tIGlzIHRoZSBiZXN0">
2  function toggleComments(id) {
3  $('#' + id + ' .commentsContainer').toggle(500);
4  }
5  </script>
```

Listing 5.5: A simple script with nonce

5.4.3 Using `Function` (Security Issue#3)

As discussed in §5.3.3, using `eval()` is dangerous and not recommended since it executes in the app's global scope and evaluates any expression passed to it as parameter exposing

hybrid companion apps to injection attacks [179]. As per the Mozilla Developer Network (MDN) Web Docs [193] and the Cordova Security Guide [225], developers should never use `eval()` in app code. The MDN Web Docs' `eval()` function documentation displays a warning regarding its usage that can be seen in Fig. 5.16.

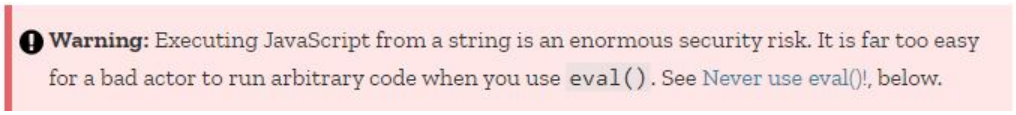

Figure 5.16: Warning against using `eval()` on MDN Web Docs

To avoid this security issue, the developer can use the JavaScript `Function` [241] object instead of using `eval()`. A `function()` definition in JavaScript is a `Function` object and can be used to evaluate expressions, similar to `eval()`. Additionally, a `function()` executes in a local scope and poses less danger to the entire app. As seen in Listing 5.6, if an attacker controls the value of `id`, they can execute malicious code in the global scope. However, in case of a `function()` (Listing 5.7), the malicious code is limited to the local scope of the function and reduces the attack surface.

```
1  eval(id + "is the employee ID")
```

Listing 5.6: Using `eval()` to evaluate an expression

```
1  function testID(id){
2      return (id + "is the employee ID")
3  }
```

Listing 5.7: Using `function()` to evaluate an expression

5.4.4 Input Validators and Output Sanitizers (Security Issue #4)

Section 5.3.4 discusses DOM and jQuery APIs and attributes that consider any input they receive as code and are vulnerable to injection attacks. In order to prevent this security issue, the developer should display app content using safe APIs (Table 5.1) that are immune to injection attacks since they consider all inputs passed to them as data [1]. However, it is common to use unsafe APIs (Table 5.1) and dynamically generate HTML elements on an app page. Therefore, while using unsafe APIs, the developer should do the following—

- *Validate input*—before processing any untrusted input, the developer should validate the input to ensure only correctly formatted data enter the app. As discussed in §5.3.4, untrusted input can be user input at client-side, and non-validated input from a database at server-side. Some examples of input validation tests include—"The input PIN should be four characters in length and consists only of numbers", "Name is a required field","Please enter the phone number in the valid format (xxx-xxx-xxxx)", "Please enter the email address in the valid format (xxxx@xxxx.com)", and "The password entered requires to be between 8 and 30 characters, contain one uppercase letter, one symbol, and a number." The developer should always validate at server-side since it is trivial to bypass client-side validation tests [248] using application level proxies, such as Burp Suite [249]. A few validator libraries that the developer can use include *Yup* [250] and *validator.js* [242].

- *Sanitize output*—before displaying any content on the app page, to prevent code injection attacks a developer should remove or *escape*—ignore the special purpose of a character or series of characters specific to a programming language [251]—any illegal characters i.e., any characters that the browser can consider as code and execute [252]. A few sample sanitizers for HTML and JavaScript include *sanitize-html* [253], *HTML Sanitizer API* [243], and *Closure-Library* [244].

5.4.5 Encrypted Storage (Security Issue #5)

As discussed in §5.3.5, commonly used storage plugins do not encrypt the data before storing it and are hence insecure. To prevent this security issue, the developer should use Cordova plugins that encrypt data before storing and securely store data in the device's local storage. A few community-developed plugins are available at the disposal of the hybrid mobile app developer, including `cordova-plugin-secure-storage` [196], `cordova-sqlcipher-adapter` [197], and `com-intel-security-cordova-plugin` [198]. The `com-intel-security-cordova-plugin` plugin is not supported by Intel anymore, however, meaning any developmental bugs in the plugin would have to be either patched by the developer or the Apache Cordova

developer's community [254].

5.4.6 Updating Cordova SDKs (Security Issue #6)

As discussed in §5.3.6, numerous apps still use old and vulnerable SDKs and expose themselves to cyberattacks. To prevent this security issue, as per the Apache Cordova team's recommendation, the developer should regularly update an app's Cordova SDKs since new versions carry important security fixes. Most updates provide critical capabilities that can improve the overall security of the app [245]. For example, Cordova version 5.1.1 introduced the capability of adding a CSP to Cordova apps.

5.4.7 Secure Allow List configuration (Security Issue #7)

As discussed in §5.3.7, Cordova apps can be configured with three types of allow lists—Navigation, Intent, and Network Request. A default or misconfigured allow list leaves the app vulnerable to a variety of attacks, such as phishing, drive-by-download, and data-exfiltration. Cordova provides a comprehensive guide on configuring all three type of allow lists to make a hybrid companion app secure [69]. The developer should note that the Cordova Allow List guide recommends using a CSP over Network Request Allow List, since a CSP allows more fine-grained control over the network requests an app can make. The Network Request Allow List still exists since older versions of `WebView` do not support CSP.

5.4.8 Secure `WebView` configuration (Security Issue #8)

As discussed in §5.3.8, insecure coding practices while implementing `WebView` can expose the app to several attacks. To prevent this security issue, while using `WebView` a developer can do the following—

- load all content over `HTTPS` by using a recognized authority's TLS [255] or SSL [255] certificate to protect user's security-sensitive information while it is in transit. The certificate also authenticates the developer's or IoT vendor's identity to ensure the user is interacting with a legitimate app;
- if the app does not require access to local file storage, disable the access;

- prevent unauthorized redirects of the `WebView` by overriding the `shouldOverrideUrlLoading()` [256] function of the `WebViewClient` class. As shown in Fig. 5.17, the `shouldOverrideUrlLoading()` function allows a developer to intercept any new URL load request in the current WebView and perform some action. This method can prevent phishing attacks by disallowing unauthorized redirects.

```
1  private class MyWVC extends WebViewClient {
2
3      @Override
4      public boolean shouldOverrideUrlLoading(WebView webView,
5                                              WebResourceRequest request){
6          if(!(Uri.parse(request.getUrl().toString())
7                                             .getHost().startsWith("legit.com"))
8              url = "https://www.legit.com/contact-us/";
9
10         webView.loadUrl(url);
11         return true;
12     }
13 }
```

Figure 5.17: `shouldOverrideUrlLoading()` to stop unauthorized redirects

5.4.9 *NOFRAK* (Security Issue #9)

As discussed in §5.3.9, since Cordova plugins reside locally on the device they have no web origin and are not protected by the SOP. Any web content, benign or malicious, loaded into the `WebView` can directly invoke the plugins to access security-sensitive information. Despite this being a major security issue, only one work in the literature, NOFRAK [37], proposes a solution for this issue. NOFRAK extends origin-based access control to the plugins by using capability tokens to authorize access to the plugins. Any web content that tries to access the Cordova plugins must be authenticated by these capability tokens. However, none of the Cordova-based frameworks implement NOFRAK and the security issue still exists.

5.4.10 No `iframe`s (Security Issue #10)

As discussed in §5.3.10, in hybrid apps content served in the `iframe` from an allow listed URL has the same privileges as the app itself and can access the Cordova plugins [225]. Therefore, if an attacker controls the served content and serves malicious code, the malicious code can access and exfiltrate sensitive device resources via plugins. To avoid this security issue, as per the Cordova Security guide, the developer should avoid using `iframe`s in app code unless they completely control the server that serves content to the `iframe` [225].

5.4.11 Updating Android SDKs (Security Issue #11)

As discussed in §5.3.11, every new Android version significantly enhances the security of the Android SDK as well [227]. Old SDKs can contain unpatched vulnerabilities and expose the app to cyber attacks. To prevent this security issue and to utilize the newer SDK's security enhancements, the developer requires explicitly declaring app support for the newer SDK through the `targetSdkVersion` attribute of `AndroidManifest.xml`. To enhance the security of the apps on Google Play Store, beginning November 2020, Google requires any new app uploads to Play Store to target Android 10 (API level 29) or higher, and any app updates to target Android 9 (API level 28) or higher [227].

5.4.12 HybridGuard

HybridGuard [56] is a policy enforcement framework that allows developers to define and enforce flexible permissions and fine-grained policies for different origins (parties) within a hybrid app. As an *inline reference monitor* [257] implemented in the web environment, HybridGuard can enforce policies at runtime to control the behavior of the JavaScript code and prevent potential attacks and resource abuses. The JavaScript code can be local script files, code from different parties such as Google AdMob for advertisements, or malicious code injected into the app via code-injection attacks. The multi-party permission and fine-grained policy enforcement mechanism advance the current "*all-or-nothing*" permission model in mobile platforms and complement basic security features provided by the hybrid framework and the embedded web browser. HybridGuard leverages JSON (JavaScript Object Notation), a textual structural specification, to design multi-party and usage control permissions. The authors implement the reference monitor and policy enforcement code in JavaScript as a single file, independent from the policy specification.

The developer can include the HybridGuard JavaScript file and the policy specification JSON into the app with minimal instrumentation of the original `index.html` page of the app. HybridGuard requires developers to mark JavaScript code from each party under a label (principal) and include it into the app using HybridGuard. By doing so, HybridGuard

can precisely monitor the code from each party and enforce policies at runtime. Using HybridGuard's policy specification the developer can define default and generic policies that can prevent frequent potential attacks such as attack scenarios discussed in §5.3.9 and §5.3.10. Since a developer can create a allow list of parties for each device resource, any injected code cannot access any Cordova plugin API because it does not belong to any party. For instance, the developer can allow `google.com` access to geolocation but restrict any other party from accessing geolocation or any other device resources. Some policy classes that HybridGuard can enforce include—

- *Volume Bound Policy*—e.g., `google.com` can read the geolocation only once a day.
- *Usage Duration Policy*—e.g., `google.com` can access the geolocation for only a minute.
- *History-based policy*—e.g., `google.com` cannot access SMS or email after accessing geolocation.
- *Location-based Policy*—e.g., `google.com` can only access location if the device is at home.
- *Allow and Block List Policy*—e.g., only `google.com` can access geolocation (allow list); `google.com` cannot access geolocation (block list).
- *Web-based policies*—e.g., creation of `iframe` is not allowed.

5.5 Conclusion

We present HybriDiagnostics, a vulnerability assessment framework that identifies eleven preexisting security issues in the companion apps of IoT devices built using hybrid mobile app development frameworks. We use HybriDiagnostics to analyze 102 real-world Cordova-based apps to identify the presence of security issues. We present an analysis of the issue conducted on our companion app dataset, results, and either a PoC attack or a synthetic attack scenario for each issue. The results show that, for several important parameters, poorly chosen defaults, improper usage of security built-ins, and improper security configurations render IoT devices

vulnerable to cyberattacks. We also provide select mitigation techniques and tools to assist IoT code producers in identifying and fixing these existent security issues.

In future work, we plan to add dataflow analysis capability to HybriDiagnostics to automatically identify if `eval()` function used in an app is vulnerable to code injection attacks. Dataflow analysis can also help in identifying whether the `String` parameters passed to the `loadURL()` function of `WebView` are HTTP URLs or JavaScript code. We also plan to add the functionality to automatically identify whether content loaded in an `iframe` belongs to a third-party and identify the third-party itself.

CHAPTER 6: CRIMINAL INVESTIGATIONS: AN INTERACTIVE EXPERIENCE TO IMPROVE STUDENT ENGAGEMENT AND ACHIEVEMENT IN CYBERSECURITY COURSES[1]

6.1 Introduction

As more organizations and governments make digital transformation a priority, the adoption of IoT technology increases. The number of IoT devices grew from 7 billion in 2018 to 31 billion in 2020 [258]. As IoT becomes widely popular, attacks become equally widespread. According to Nokia's threat intelligence report, internet-connected, or IoT, devices now make up roughly 33% of all the infected devices [259]. With the increasing number of attacks related to IoT devices, IoT security education gains importance for awareness and improving the workforce. There are several gaps to be filled in advanced cybersecurity education in order to strengthen the nation's cybersecurity workforce [49]. For example, there is a severe lack of gender and ethnic diversity in the cybersecurity industry, something that is desperately needed to meet the growing demand and foster innovation and creativity in problem solving [50, 51, 52, 53]. Addressing the above security issues requires delivering IoT security educational content engagingly and inclusively.

Prior work suggests that *gamification* [260], the application of game-design elements and game principles in non-game contexts, in classroom activities, is likely to increase student engagement and enhance learning [104]. Games in cybersecurity education enhance engagement, promote active learning in delivering education content, inspire interest in computer security, and motivate participants to explore further the field (cf., [102]). To our knowledge, Criminal Investigations is the first framework that incorporates gamification principles, universal design, and inclusivity to teach and assess advanced IoT software security topics.

[1]This chapter includes joint work with Pooja Murarisetty, Diep Nguyen, Julio Bahamon, Harini Ramaprasad, and Meera Sridhar

In this paper, we introduce *Criminal Investigations*, a gamified, scalable web-based framework for teaching and assessing *Internet-of-Things* (IoT) security skills. We envision Criminal Investigations as a consolidated package of stackable IoT security activities, each activity teaching students skills critical for the next. Starting with an introduction to essential IoT firmware components through an IoT firmware reverse engineering and analysis activity, Criminal Investigations will span activities related to vulnerability discovery and trivial and advanced firmware attacks. We present a prototype Criminal Investigations, with a fully-deployed first activity "Reverse Engineering and Analyzing IoT Firmware".

Criminal Investigations features several game design and development principles, including: (i) a narrative or story, (ii) *knowledge checkpoints* [261], (iii) rewards such as *experience points* (XP) [107, 262], and (iv) challenge [108]. Criminal Investigations includes a *Practice Mode* to allow students to solve ungraded module challenges and a *Test* mode that presents more difficult challenges and a graded quiz. Criminal Investigations also reinforces key concepts via *just-in-time* learning content delivery while the student is engaged in the activity. Criminal Investigations uses React [6] for the front-end and Python for the back-end and is deployed as a web application on Amazon Web Services (AWS) cloud. For the "Reverse Engineering and Analyzing IoT Firmware" activity, we provide the student with an IoT firmware image and a virtual environment (virtual machine image) with the necessary analysis tools pre-installed. The goal of the activity is to reverse-engineer the firmware using the tool `binwalk` [263], and identify information such as the type and version of the firmware kernel, the type and version of the firmware bootloader, compression schemes used, the hardware architecture, and others. Identifying this information is the foundation of firmware security analysis. It is used in decompressing firmware data, identifying pre-existing vulnerabilities, and creating *proof-of-concept* exploits to demonstrate the consequences of the vulnerabilities.

We design the "Reverse Engineering and Analyzing IoT Firmware" activity as a narrative featuring a detective and a college professor, addressed to a student (who is completing the activity) from the cybersecurity department regarding an ongoing investigation of compro-

mised IoT devices on campus. As part of the investigation, the campus police has seized a suspect's laptop in the case. The narrative leads the student helping the detectives through the analysis of the firmware files found on the laptop, which will help identify details about the compromised devices. Before beginning the activity, the student must read the learning content and pass a Knowledge Checkpoint quiz that assesses the student's preparation to attempt the activity. Auditing the learning content and reaching the Knowledge Checkpoint is critical since the information from the readings is required to solve the activity challenges. We report on preliminary feedback on Criminal Investigations, obtained through a small-scale pilot study.

The main contributions of this paper are—

- the design, development, and deployment of Criminal Investigations, a gamified, scalable web-based framework for teaching and assessing IoT firmware security skills;
- Criminal Investigations's fully deployed first activity "Reverse Engineering and Analyzing IoT Firmware";
- results from a small-scale pilot study that obtains feedback on the benefits of Criminal Investigations, including increased engagement, learning, and excitement.

Roadmap Section 6.2 presents the pedagogical goals and strategies that support the design of Criminal Investigations. Section 6.3 discusses the high-level design of Criminal Investigations. Section 6.4 outlines details about the prototype activity. Section 6.5 discusses the development and deployment of the framework and the first activity. Section 6.6 presents the feedback received from the small-scale pilot study. Section 2.5 briefly discusses related work. Section 6.7 presents our conclusions and discusses future work.

6.2 Pedagogical Goals and Strategies

We now outline our primary educational goals and present the strategies that we employ to meet these goals.

6.2.1 Educational Goals

G1: Promote student learning and engagement. The role of a teacher is no longer that of the primary source of *information*, rather a *facilitator* who helps students develop and hone higher-order cognitive skills [112]. The focus is on engaging students in discussions or activities, helping them think critically, and enabling them to be lifelong learners. We aim to incorporate strategies to improve student learning and their engagement with the course material, each other, and instructors.

G2: Motivate students to explore advanced topics in cybersecurity. The threat of cyberattacks to national security is real, and currently, there is a national shortage of skilled cybersecurity workforce [50, 49]. We aim to motivate students to develop an interest in advanced cybersecurity topics such as IoT security and maintain and grow this interest in the years ahead. Such an interest can potentially lead them into a successful career in a field that is always going to be in high demand.

G3: Promote inclusivity, accessibility and broader dissemination. Bringing multiple perspectives through a diverse workforce is a driving force to inspire creativity and innovation in a field like cybersecurity [52] where new types of security vulnerabilities and attacks arise all too frequently. There is an unfortunate lack of diversity in the cybersecurity workforce [53] and the cybersecurity higher-education pipeline. We aim to make advanced cybersecurity topics accessible to a diverse and broad body of students.

6.2.2 Strategies to achieve Educational Goals

Our primary strategy to increase student engagement and learning (**G1**) and to motivate students to explore advanced topics in cybersecurity (**G2**) is to employ an interactive, gamified approach to teach and assess IoT security. Past works show that gamification increases student engagement and motivation [104, 106].

To promote inclusivity, accessibility, and broader dissemination (**G3**), we (1) design Criminal Investigations as a web-based application that is available online and easily accessible through any web browser; (2) incorporate diverse examples and avoid stereotypes that are

prevalent in the field of cybersecurity within our narrative; (3) make our best attempt to adhere to guidelines for universal design in our user interface (e.g., choice of font sizes, styles and color scheme).

6.3 Design

The key idea behind the design of Criminal Investigations is to promote student learning and engagement in topics related to IoT security by incorporating elements of gamification [104] into hands-on activities. Criminal Investigations presents activity in the form of a narrative (Fig. 6.1) to improve student engagement and incorporates knowledge checkpoints (Fig. 6.2) to assess student preparedness for the activity. Criminal Investigations awards eXperience Points (XP) to students at various checkpoints throughout the activity to keep them motivated. Criminal Investigations features *just-in-time* learning content delivery to reinforce key concepts during an ongoing activity. Criminal Investigations also has a Practice Mode to provide students with opportunities to sharpen the knowledge and skills required to complete the activities successfully. We provide students with a virtual machine image, pre-packaged with all software and tools required to solve challenges in a particular hands-on activity. For ease of accessibility, we deploy Criminal Investigations as a browser-based framework hosted in a cloud environment.

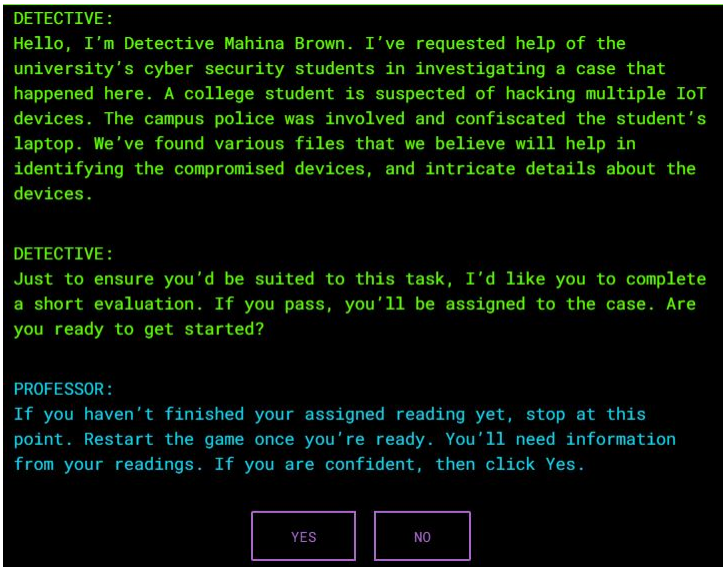

Figure 6.1: Activity as a narrative

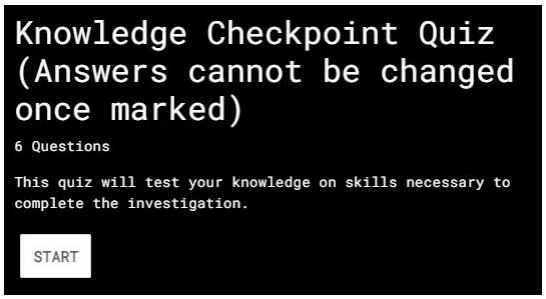

Figure 6.2: Knowledge Checkpoint in Criminal Investigations

6.3.1 Activity Gamification

For activities, we refer to previous works that establish the success of gamification concepts in Computer Science education [106, 104]. The key idea behind gamification is to understand which mechanics keep gamers motivated to come back to play and apply those constructs to non-game environments to encourage similar engagement. The goals in designing Criminal Investigations are to increase student engagement in IoT security education while also making the content accessible to a diverse audience. Based on prior research that establishes *interaction* as an essential element in making games and activities engaging, a key focus in our design is to ensure interactivity [108, 107]. We achieve this by transforming a traditional course assignment into a narrative-based interactive activity that incorporates gamification concepts such as experience points (XP) and checkpoints.

6.3.2 Game Modes

As shown in Fig. 6.3, the framework supports two separate modes—Practice and Test.

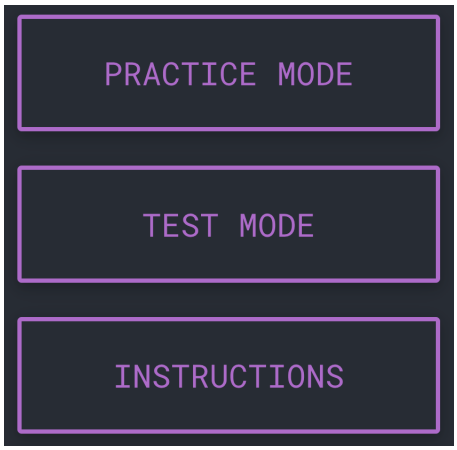

Figure 6.3: Main screen for Criminal Investigations

6.3.2.1 Practice Mode

The purpose of the Practice Mode is to allow students to get accustomed to the activity's environment and practice the skills required to complete the activity, with inputs or configurations chosen specifically for practice mode. Students have unlimited attempts in the Practice Mode. However, Practice Mode does not allow the student to save progress. Therefore, any page refresh ends in progress loss.

6.3.2.2 Test Mode

The student uses the Test Mode to complete an activity as part of a *g*raded assessment within the course. The Test Mode can be configured to limit students to a specific number of attempts (e.g., two attempts) for the activity as desired by the instructor. Since the Test Mode has a limited number of attempts, a student first requires to *Knowledge Checkpoint* quiz to ensure that they are adequately prepared to start the activity. The Knowledge Checkpoint has a minimum point (XP) threshold that the student must achieve before progressing to the activity.

6.3.3 Just-in-Time learning content delivery

Criminal Investigations is envisioned to complement learning content such as lecture videos, readings, and tutorials rather than being a replacement. As seen in Fig 6.4, we incorporate snippets of learning content right into the narrative and activity to reinforce key concepts.

6.3.4 Ease of Access

Criminal Investigations is an interactive web-based application developed using React JS [6] for the user interface or front-end, Python Flask [264] library for the backend, and MongoDB [265] as the backend database. We provide all the tools and files required to complete a given activity as part of a pre-built virtual machine (VM) image. Criminal Investigations is easily accessible from any web browser, with the landing screen shown in Fig. 6.5.

Upon clicking START, the student is prompted to enter their ID and password. Once Criminal Investigations verifies the student's enrollment in the course, the student can access

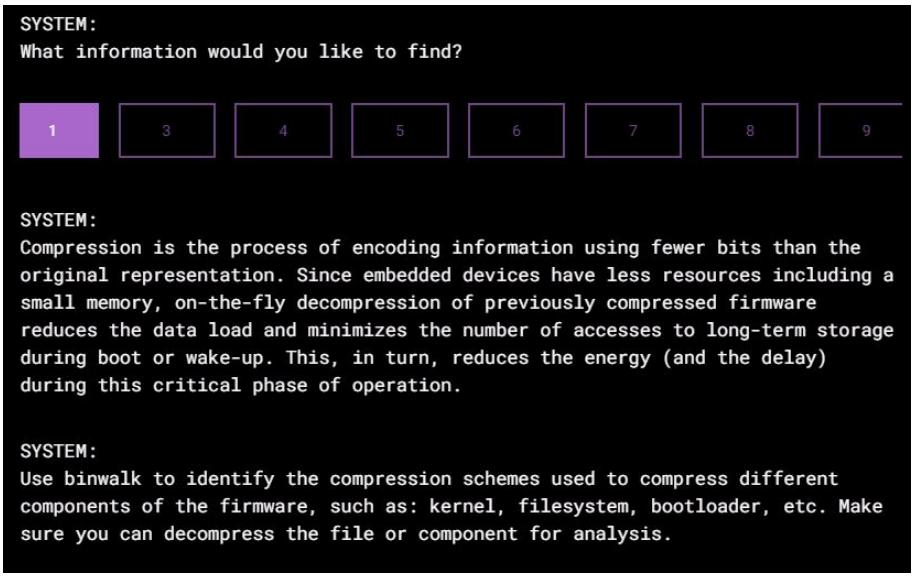

Figure 6.4: Just-in-Time learning

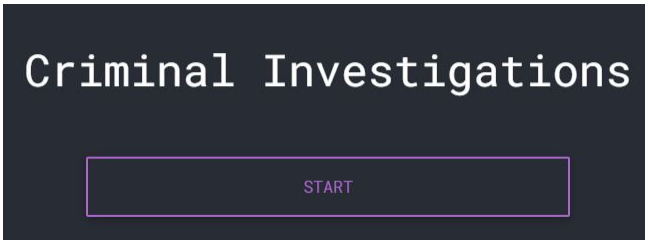

Figure 6.5: Landing screen for Criminal Investigations

the application's main screen, shown in Fig. 6.3.

6.4 Prototype Activity: Reverse Engineering and Analyzing IoT Firmware

In this activity, the student's goal is to reverse engineer an IoT firmware image using `binwalk` [263] and extract and identify various components of the firmware. The student must identify firmware components that include compression schemes used for the filesystem or elsewhere, kernel, bootloader, filesystem, user apps, web apps, and CPU—endianness, architecture, and processor type (32-bit/64-bit). Identifying these components is critical for further analyzing the firmware image and diagnosing pre-existing security issues. For example, if the firmware uses an outdated kernel or bootloader containing pre-existing vulnerabilities, an attacker can exploit these vulnerabilities to hijack the IoT device. Information such as CPU architecture and its type and endianness assists in constructing *proof-of-concept* exploits since every architecture type has a different set of instructions, opcodes syntax, count, and

types of registers.

In this activity, we incorporate the various design features that we discussed in Section 6.3:

Narrative Style As seen in Fig 6.1, the activity begins with an introductory narrative featuring a detective and a college professor, addressed to a student (who is completing the activity), regarding an ongoing investigation of compromised IoT devices on campus. An unknown entity has compromised specific university IoT devices, and as part of the investigation, the campus police have seized the laptop of a suspect in the case. The laptop contains firmware files that the police believe are from the compromised IoT devices, and the cybersecurity department has to assist the police department in analyzing the files. Once the introductory dialog ends, the student can choose to proceed or come back later to begin the core activity.

Practice and Test Mode We provide the student with the firmware image and the necessary tools to complete the activity. The student can access the prototype activity in both the Practice and Test Modes. As mentioned earlier, the Practice Mode allows the student to familiarize themselves with the activity environment and the tools and files provided. The Test Mode starts with a Knowledge Checkpoint quiz. Once the student achieves a pre-defined threshold in this quiz, they can access the core activity components, where they are required to reverse engineer and analyze the assigned firmware and answer questions based on the analysis to help solve the case.

Reward System To keep the student motivated throughout the activity, Criminal Investigations provides instant feedback in the form of encouraging dialog and XP for correct answers.

Activity Requirements As seen in Fig. 6.6, the student needs to complete nine activity tasks, one at a time. The tasks are non-sequential and accompanied by a small summary and security relevance of the requirement. To fulfill a requirement, the student must perform a particular analysis task, such as finding the compression scheme used to compress the

firmware file system and answer an analysis-based question.

Virtual Environment We provide the student with a virtual machine (VM) image that has `binwalk` and its dependencies pre-installed and accessible from the terminal. For the pilot study, we thoroughly test the VM after installing `binwalk` before exporting it using the Open Virtualization Format (OVF) [266]. We also create a *snapshot* [267] of the VM to restore the system to its original state with `binwalk` installed. The VM image was accessible through Google Drive. The size of the associated Virtual Machine Disk (VMDK) was 3.76 Gigabytes.

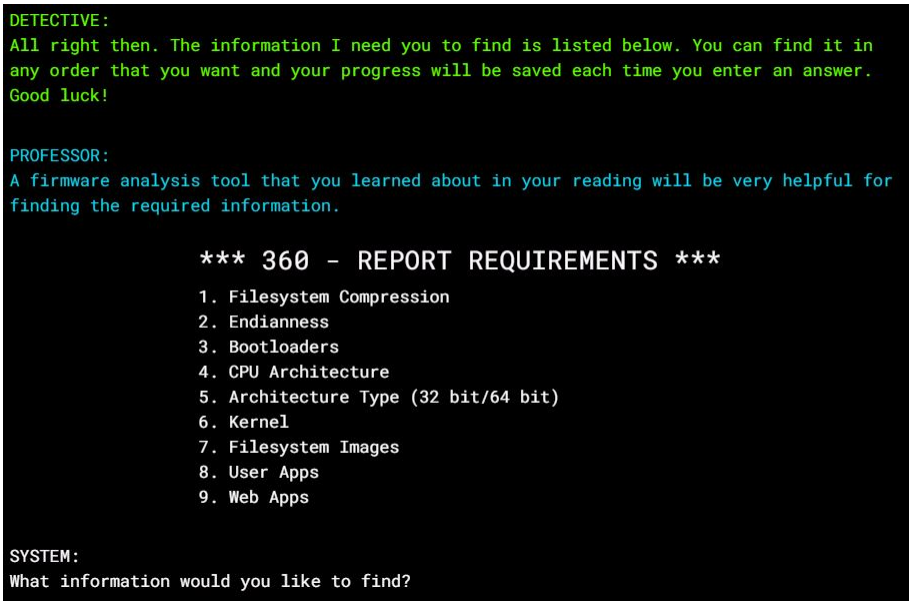

Figure 6.6: A sample view of Criminal Investigations

6.5 Implementation and Deployment

Criminal Investigations's implementation and deployment included three major aspects:

1. Designing and developing the front-end, i.e., user interface (UI) for the framework using React JS (open-source JavaScript library) [6].

2. Designing and developing the backend using Python's Flask library [264] in combination with MongoDB [265] for the database.

3. Deploying a prototype of Criminal Investigations to Amazon Web Services (AWS) to conduct a pilot study.

6.5.1 Implementation

Front-end We implement the UI of Criminal Investigations using React JS [6], a JavaScript library for building responsive and stateful UI components. React follows a component-based approach to provide modularity and re-usability. We develop components of a web page, such as header, navigation bar, sidebar, footer, and others, individually and then combined them to form different views. Instead of following the traditional concept of a multi-page web app, we use simple views for each state in the activity, and React efficiently updates only the required components when the data changes.

Quiz component For skill assessment, Criminal Investigations incorporates a multi-question Knowledge Checkpoint quiz and a single-question quiz as part of each of the nine requirements. To create the quizzes, we used the *react-quiz-component* [268], an open-source React component that simulates a simple quiz engine. We modify the quiz component to suit our requirements. We store the quizzes as JavaScript Object Notation (JSON) objects [153].

Database Since we store the quizzes and narrative dialogs as JSON objects, we chose MongoDB [265] as our datastore due to the ease of storing and retrieving JSON objects from MongoDB.

Backend We developed our framework's backend using the Python Flask library [264]. Currently, the backend is responsible for packaging and delivering the UI and connecting with the MongoDB database. Additionally, we use the backend for storing and retrieving student data from the database and verify the student's enrollment in the course before allowing them to access the activity.

Accessibility The UI design follows university design and accessibility guidelines to allow users of diverse abilities to navigate, understand, and use the UI. There is a high color contrast ratio between the colors used in the UI for better readability. The layout and typography are also compliant with accessibility principles. There is a deliberate delay when rendering dialogs to help the student easily follow the story, accompanied by a default scroll to the

bottom to make it easier for the student to find the location of the current task.

Engagement and Motivation As seen in Fig. 6.7, both the Test Mode and Practice Mode have a status bar at the top that displays the amount of XP the student has acquired. To keep the student motivated, the XP updates immediately after the student completes a task. After completing each requirement, the student receives a congratulatory message to reinforce the gamification principle of rewards.

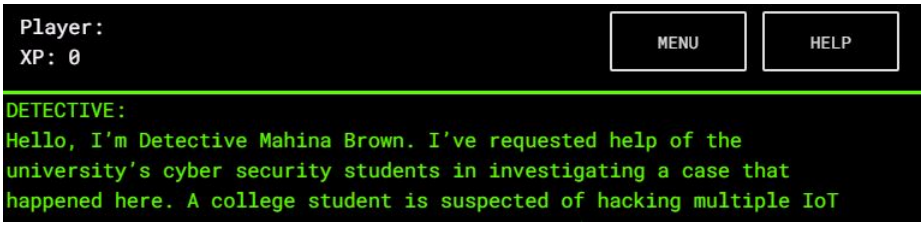

Figure 6.7: Test Mode for Criminal Investigations

6.5.2 Deployment

We deploy our current prototype of Criminal Investigations on Amazon Web Services (AWS) *t2.micro EC2* machine that runs Ubuntu 18.04 LTS. Criminal Investigations is accessible via a temporary domain (not disclosed for anonymity). We deploy the React front-end and the Flask backend as two micro-services using Docker containers. We use NGINX, an open-source web server, which besides serving the static files of Criminal Investigations, also listens to HTTP/HTTPS incoming traffic and transfers client requests to the backend service. It also redirects the traffic from HTTP to HTTPS.

6.6 Pilot Study

We conducted a small-scale pilot study with a group of ten Computer Science students to obtain preliminary reactions and feedback on our Criminal Investigations prototype. We Students were asked to complete a course module on Reverse Engineering and Analyzing IoT Firmware, which included short pre- and post-surveys, learning content, the gamified activity detailed in Section 6.4, a final quiz and an anonymous student feedback survey specifically to get feedback on the activity and the framework design. A few of the students indicated that they had a little prior experience in firmware security or security in general, but most

indicated that they did not.

The student survey asked for feedback and suggestions on the following—

- User Interface accessibility—color contrast, text size, layout, navigation;
- Narrative and instructions—clarity of dialogs and instructions, speed between dialogs within the narrative, clarity of game-play rules;
- User experience—time spent on completing the setup and activity, whether the activity was engaging, whether the narrative and XP motivated them to do well in the activity and whether the activity helped reinforce the topics covered in the module.

While all participants completed the course module, activity, and quizzes, only seven out of the ten participants responded to the anonymous survey. We summarize the overall reactions and feedback below.

User Interface accessibility A majority of the respondents indicated that they were either very satisfied or satisfied with the User Interface accessibility aspects of color contrast, text size and navigation. However, some respondents expressed the need for improvement in the layout (i.e., placement of some of our interactive elements).

Narrative and instructions Respondents indicated satisfaction with the clarity of game-play rules and the dialogs and instructions within the narrative itself. However, some indicated the need for more precise instructions for the setup of the virtual environment needed for the prototype activity.

User experience A majority of the respondents were able to complete the module and activity within the expected time of 1 to 2 hours. However, some students took longer due to installation issues. All respondents indicated that the activity was engaging, and they were motivated by XP (or just the ability to earn points in general). All except one respondent also indicated that doing the activity reinforced the learning content introduced in our course module.

Overall, the feedback we obtained from the pilot study is very encouraging and gives us valuable suggestions for improvement of the Criminal Investigations framework.

6.7 Conclusion and Future Work

In this work, we present Criminal Investigations, an interactive, gamified framework for teaching and assessing IoT security skills. Our goal is to provide the students with a more enjoyable and engaging environment to learn these skills. Our prototype implementation of Criminal Investigations features an introductory "Reverse Engineering and Analyzing IoT Firmware" activity. The results of a small-scale pilot study indicate that the framework is engaging and accessible.

In the Spring 2021 semester, we plan to deploy our prototype Criminal Investigations activity (after addressing and incorporating a few suggestions that we received from the small-scale pilot study) in multiple sections of a junior level undergraduate course teaching Operating Systems and Networking concepts (reaching approximately 300 students) and in a section of a junior or senior-level introductory Game Design and Development course (reaching approximately 75 students).

In future research and development, we plan to add several enhancements to our framework, including activities with increasing levels of complexity and progression requirements, the ability for students to earn incentives and unlock challenge levels based on earned XP, and increase in randomization and adaptivity of the activities using concepts of Artificial Intelligence.

CHAPTER 7: CONCLUSION

In this dissertation, we propose a robust and comprehensive security solution for securing hybrid mobile apps. Our dissertation consists of three main thrusts—(i) policy enforcement—our policy enforcement framework can enforce fine-grained policies in hybrid apps to protect against attacks that originate from third-party JavaScript included by the developer and code-injection attacks; (ii) vulnerability assessment—our vulnerability assessment framework can identify subtle security issues in hybrid companion apps at the development stage itself; and (iii) cybersecurity education framework—our cybersecurity education framework can be used to teach the concepts of hybrid app and IoT firmware security and assist in strengthening the foundation of the cybersecurity workforce.

7.1 Hybridguard: A multi-party, fine-grained permission and policy enforcement framework for hybrid mobile applications

In Chapter 3 and Chapter 4, we present HybridGuard, a principal-based, fine-grained policy enforcement framework for hybrid mobile apps that allows developers to enforce stateful policies to mitigate attacks originating from third-party JavaScript code and code-injection attacks. HybridGuard is platform agnostic; therefore, it can be deployed for apps developed using hybrid app platforms on both Android and iOS. HybridGuard can also enforce a broad class of policies that the app developer can use to mitigate attacks that can breach a user's privacy and exploit the smartphone. We thoroughly evaluate HybridGuard using real-world hybrid apps and evaluate its compatibility with various hybrid app development frameworks. We ensure the integrity of HybridGuard by enclosing the implementation in an anonymous JavaScript function. We protect the integrity of the policy specification by prohibiting unauthorized access, enforcing it with HybridGuard's monitor. We also ensure complete mediation of security-sensitive APIs by systematically exploring and mediating all possible aliases and channels that can generate dynamic JavaScript code. Finally, we also

provide the end-user with the capability to customize the policies to their requirements.

7.2 HybriDiagnostics: An automated vulnerability assessment framework for hybrid smart home companion apps

In Chapter 5, we present HybriDiagnostics, an automated security assessment tool for hybrid companion apps that can assist developers in mitigating subtle preexisting security issues that can lead to user privacy compromise in smart home ecosystems. HybriDiagnostics can identify misconfigured policies (including Content Security Policy and whitelist), usage of inline scripts, unsafe `eval()` usage, unsafe HTML and JQuery APIs and attributes, unencrypted storage, usage of vulnerable Cordova SDKs, and others. We evaluate HybriDiagnostics using 102 real-world Cordova-based smart home companion apps and present our results. For each security issue, we also provide a PoC attack or a synthetic attack scenario to show how each security issue can be exploited in a smart home ecosystem and its consequences. We also provide select mitigation techniques and tools that the developer can use to mitigation the security issues and reduce the attack surface.

7.3 Criminal investigations: An interactive experience to improve student engagement and achievement in cybersecurity courses

In Chapter 6, we present Criminal Investigations, an interactive, gamified, scalable web-based framework for teaching and assessing cybersecurity skills. Criminal Investigations provides students with a more engaging and enjoyable environment to learn cybersecurity skills. Criminal Investigations's prototype implementation features an introductory "Reverse Engineering and Analyzing IoT Firmware" activity. We envision Criminal Investigations as a set of stackable activities that teach basic and advanced cybersecurity skills. We plan to augment Criminal Investigations with a series of activities including topics from hybrid app security, IoT firmware security, and others. We also present the results of a small-scale pilot study that indicates the framework is engaging and accessible.

7.4 Summary

Here we summarize the answers to the research questions mentioned in §1.1.

1. What types of cyberattacks can originate from the inclusion of third-party JavaScript

in hybrid mobile apps? **[Chapter 3]**

The in-scope threats originate from third-party JavaScript code included from a source (domain) allow listed in the CSP. The third-party JavaScript code could be benign but under the control of an attacker through web application attacks, such as SQL injection, or a network attack on the third-party server; malicious by intentions. Some attacks include abusing device resources, sensitive information leakage through malvertisement, overuse of resources, and UI redress attacks. In §3.2.1 we provide detailed scenarios of attacks that can originate from third-party JavaScript included in a hybrid app by the developer.

2. To what extent do security mechanisms built into the mobile OS, or provided by the embedded browser, or provided by the hybrid app frameworks provide security for hybrid apps from cyber attacks originating from the inclusion of third-party content in hybrid apps? **[Chapter 2]**

   The existing security enforcement mechanisms for hybrid mobile apps consist of an inadvertently patched together model, with separate security for the native and web components. Attackers exploit gaps in this model, and abuse bridge code to access device resources. Hybrid mobile apps use the same permission model as the native OS to allow access to device resources. Users can grant permissions at run-time to access device resource, such as geolocation, Email, and so on. However, once the permission has been granted, there is no way to control how the app uses these permissions. Unlike traditional web apps, in hybrid mobile apps the origin of the JS code is not propagated and allows any JS code included in the app to access any device resource the app has permission to access. This also makes it impossible to enforce policies based on the real origin of the API invocations.

   Some hybrid frameworks provide plugins to implement an allow list of domains that can be accessed from the app. However, scripts included by the developer, such as ad scripts,

have to be allow listed and can become malicious at a later stage. Content security policy is another native browser capability that gives the developer more fine-grained control what content and what domains an app can access. But this also has the same drawback as domain allow listing. Developer added scripts still need to be added to the CSP. Same Origin Policy is also enforced by the browser, and limits interaction between JS code and the app based on the origin. It also does not work in hybrid apps as the JS bridges are added to the browser by local code and have no web origin as far as the browser is concerned. This allows malicious web content to directly invoke these bridges.

Our research shows that the built-in security mechanisms do not provide sufficient security and can be trivially bypassed by an attacker as demonstrated in §3.2.1.

3. What are the most prevalent security issues in hybrid mobile apps? **[Chapter 5]**

   Apart from malicious third-party JavaScript, we identify eleven security issues that are prevalent in hybrid mobile apps. To identify these security issues we use the results of our research, and read numerous research papers, blog posts, and articles related to hybrid mobile app security. Specifically—

   (a) our analysis of hybrid apps while designing HybridGuard (policy enforcement framework) reveals security issues such as broken SOP, missing CSP, and usage of inline scripts;

   (b) we refer to the Cordova Security Guide and security advisories by the Cordova team to identify a few security issues such as usage of `iframe`s, misconfigured allow lists, and unencrypted storage;

   (c) by reading numerous blog posts and research papers we select security issues such as outdated libraries and SDKs, unsafe DOM APIs, and `WebView`-based attacks.

   Once we identify the most prevalent security issues, we analyze a dataset of 102 real-world smart home companion apps to identify the presence of these pre-existing security

issues. The result of our analyses is as follows:

(a) Content Security Policy—out of the 102 Cordova-based apps, only 32 apps implement a CSP, implying that developers of 70 apps chose to delete the CSP. Out of the 32 apps that implement a CSP, ten have a default CSP, and 22 apps have a misconfigured CSP.

(b) Usage of inline JavaScript—out of 102 Cordova-based apps in our dataset, 71 apps use inline JavaScript.

(c) Usage of `eval()` in app code—out of the 102 Cordova-based apps in our dataset, 50 apps use eval() in the app code.

(d) Usage of unsafe DOM APIs—out of 102 Cordova-based apps in our dataset, 84 apps use unsafe APIs to display the app's content.

(e) Usage of unencrypted storage—out of the 102 apps in our dataset, 92 apps use insecure APIs and plugins for storing data.

(f) Usage of vulnerable Cordova SDKs—out of the 102 Cordova-based apps, 38 apps use vulnerable SDKs.

(g) Default or misconfigured allow list—out of the 102 Cordova-based apps, 43 apps have a default allow list, and 58 apps have a misconfigured allow list.

(h) `WebView` Attacks—Out of 2082 apps in our entire dataset, 1019 apps use `WebView`; 125 apps use `WebView` to load local content; 565 apps use `loadUrl()` function of the `WebView` class to execute JavaScript; 808 apps have JavaScript enabled in `WebView`; 111 apps load content over http; and 232 apps that enable `WebView`'s access to local storage of the device.

Additional details are provided in §5.2.4 of the dissertation.

4. What are the challenges of designing a secure IRM framework in the cross-domain platform (HTML, CSS, JavaScript) of hybrid mobile apps? **[Chapter 3]**

There are three main challenges that we need to overcome to design a secure IRM framework:

(a) Complete Mediation—our framework should be able to identify and intercept all security relevant events, such as calls to any security-sensitive API. We achieve this by wrapping all the security-sensitive APIs and ensuring that the original APIs can only be accessed through our wrapper APIs.

(b) Attribution—our framework should be able to accurately identify the origin of the security-sensitive event. In the case of a call to a security-sensitive API, our framework should be able to identify the calling party or origin. We achieve this by loading any external JavaScript using our framework's API rather than using the `<script>` tag, and labeling each external JavaScript. During any call to a security-sensitive API, the corresponding label (also known as principal) of the external JavaScript is pushed onto a local shadow stack. Then the policy engine determines if the calling external JavaScript is allowed to access the security-sensitive API. Once the execution is over, the principal or the label is popped off the stack.

(c) Tamper Proofing—our framework should be able to maintain its own integrity and should be resistant to any tampering by the attackers. We ensure tamper proofing by designing our framework as a single JavaScript file and leveraging the concept of lexical scoping in JavaScript, i.e., enclosing the entire code in an anonymous function. This prevents any external content from being able to access our framework's code, hence, maintaining its integrity.

Additional details can be found in §3.3 and §3.4 of the dissertation.

5. Can in-lined reference monitoring provide an elegant solution for protecting against attacks on user's privacy in hybrid mobile apps (especially privacy attacks originating from third-party JavaScript)? **[Chapter 3 and 4]**

Our experiments demonstrate that our IRM framework is capable of providing retroactive protection against attacks originating from third-party JavaScript by enforcing principal-based fine-grained policies. Our IRM framework provides protection without modifying the underlying operating system or the hybrid app development frameworks. Compared to traditional reference monitors that reside outside the program to be monitored, an in-lined reference monitor resides inside the untrusted code and has access to all program states and requires less context switching. We design our framework as a single JavaScript file that can be added to the app code and packaged with the APK to provide runtime protection. We evaluate our IRM framework based on three critical aspects of security described in the question above.

Our IRM framework extends the OS permission model that can only allow/disallow access to a resource by introducing access qualifiers such as read, write and create. Using IRMs we can enforce multi-principal policies for each resource. An example policy that can be enforced is to "*disallow access to* `SMS.send()` *API if an untrusted principal has accessed geolocation*". To evaluate the compatibility and usability of our framework, we integrate our framework into existing real-world hybrid apps. To evaluate whether our framework can soundly enforce these policies, our test policies not only log API execution but also monitor the behavior of the execution with context to the fine-grained policies as provided as templates presented in §4.4. These policy templates include multi-party and context-aware permissions in the JSON specification that can prevent the attack scenarios of abusing device resources, as identified in §3.2.1. Some other examples of policies include:

- *allow access to geolocation only while at home* (implemented using latitude and longitude coordinates);
- *allow access to geolocation only for fifteen minutes*;
- *allow sending of SMS to only specific contacts from the contact list*; and

- *disallow creation of invisible* `iframe`*s.*

6. What are the classes of security policies that can be enforced by such an IRM framework? **[Chapter 3 and 4]**

   Our IRM framework is capable of enforcing a wide class of fine-grained security policies. Some of the policy classes that can be enforced by HybridGuard include:

   (a) Volume Bound Policy—restrict access to a resource to a specific number. We also provide an additional time unit parameter with this policy that specifies the amount of time until the count resets. For example, the geolocation API can only be accessed 10 times in a day.

   (b) Duration Usage Policy—allow access to a resource only for a particular amount of time. For example, geolocation API can only be accessed for an hour.

   (c) Location-based Policy—allow access to a resource only from a particular location. For example, geolocation API can only be accessed from the user's home coordinates.

   (d) History-based Policy—policies that take into account the sequence of events. For example, once the geolocation API is accessed, the internet cannot be accessed by the calling party.

   (e) Block Lists and Allow Lists—a specific set of principals can either access a resource or cannot access it.

   (f) Web-Security Policies—restrict the creation of certain DOM elements. For example, restrict creation of invisible `iframe`s to prevent phishing and clickjacking attacks.

   (g) Custom fine-grained policies—developers can also create custom policies by modifying/using the provided templates. For example, once a third-party script accesses the gallery, it should not be able to access any device resource that can be used to exfiltrate user's images or videos.

Additional information about policy patterns and templates can be found in §4.4 of the dissertation.

7. How should the policy specification language or platform be designed to also allow users to define the policy? **[Chapter 4]**

   We design the policy specification as a key-value pair, and choose JSON for the policy specification. We choose JSON since its lightweight and compatible with JavaScript. In the initial design of our IRM framework we store the policy specification as part of the app code, i.e., the JSON policy specification file is stored as part of the APK. However, after carefully rethinking the IRM design, we now store the JSON policy specification file in the device's local storage instead of storing it as part of the app code. This design modification allows the users to customize the policies at their end. Additional details about the policy specification and storage can be found in §4.3 of the dissertation.

8. What is the impact on performance of an app after integrating the IRM framework and enforcing policies? **[Chapter 4]**

   We test the impact on performance on both Android and iOS. We evaluate our framework performance by measuring the runtime overhead posed by our policy enforcement mechanism. We measure the load time of an app with and without our framework. We do not notice any slowdown as the load time of the original app and the modified app with our framework are identical. This result can be explained by the fact that JavaScript code in hybrid apps is mostly event-based, and asynchronous. For this reason, we evaluate micro-benchmarks of operations that do not depend on triggered events, including getting the current position, acceleration, and direction. We modify the code in original app variants to execute these operations 1000 runs, to achieve high precision, and measure the time before and after the runs. For each case, we run the apps on the two devices (Android and iOS) with ten trials to get the averaged numbers. The details about the performance evaluation and results can be found in §4.5.3 of the

dissertation.

9. Which hybrid app development frameworks should we target the IRM framework to be compatible with? **[Chapter 4]**

   We evaluate our IRM framework for compatibility with Apache Cordova, Phonegap, Ionic, Framework7, Onsen UI, and Intel XDK. Our framework is compatible with all these hybrid app development frameworks. For the evaluation, we develop a test suite of apps using different hybrid app development frameworks, and also evaluate using real-world apps. Additional details can be found in §4.5.1 of the dissertation.

10. How can security issues in smarthome companion hybrid mobile apps be exploited to attack a smart home ecosystem? **[Chapter 5]**

    With the help of proof-of-concept attacks and synthetic attack scenarios we demonstrate how these existing security issues can be exploited to launch serious cyber attacks.

    (a) Stealing sensitive user data such as location information, photos, contacts, and other security-sensitive data.

    (b) Disabling alarm systems in a smart home.

    (c) Changing temperature of a thermostat.

    (d) Disable IP cameras

    (e) Access unencrypted storage to steal sensitive API keys to send commands to smarthome devices

    (f) Cause denial of service by injecting splash screens, pop ups and crashing the app

    (g) Phishing attack to steal login credentials

    (h) Switch on the irrigation system without the user's knowledge and cause monetary damage

11. Does gamification help in improving student engagement and learning in advanced cybersecurity topics? **[Chapter 6]**

- Does using a narrative increases the student's interest in the activity and capture their attention?
- Does earning experience points (XP) for solving activity challenges motivate the student to perform well in the activity?
- Does the design of the activity, i.e., colors, fonts, and placement of UI elements follow accessibility principles?

We design a text-based gamified activity to teach and assess reverse-engineering and firmware analysis skills in upper-division undergraduate cybersecurity courses. The activity incorporates elements of game design such as storytelling, experience points (XP) and just-in-time learning content delivery to increase student engagement, interaction, and learning. The activity is implemented as an easily accessible web-based application, deployed in a cloud-based environment. We conducted an initial small pilot study of ten students to get feedback on some of the gamification elements incorporated in the activity and also about the over all activity. We received positive feedback from the students in terms of increasing engagement and interaction. We also received some critical feedback to improve the activity instructions, and placement of some of the UI elements. We incorporated‘ the feedback and conduct another medium-sized study of 300 students. Our study shows that an interactive and gamified framework can increase engagement and interest of students in learning advanced cybersecurity skills.

REFERENCES

[1] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proc. of the 2014 ACM SIGSAC Conf. on Comp. and Comm. Sec. (CCS)*, pp. 66–77, 2014.

[2] P. Saccomani, "Native Apps, Web Apps or Hybrid Apps? What's the Difference?." https://www.mobiloud.com/blog/native-web-or-hybrid-apps. Accessed on 04-20-2021.

[3] S. Nunez, "Your phone is now more powerful than your PC." https://insights.samsung.com/2020/08/07/your-phone-is-now-more-powerful-than-your-pc-2/, 2020. Accessed on 04-20-2021.

[4] P. Peranzo, "App development decisions: Native app vs web app vs hybrid?." https://www.imaginovation.net/blog/app-development-decisions-native-web-or-hybrid/, 2018. Accessed on 04-20-2021.

[5] M. Butusov, "Native vs Hybrid apps. What to choose in 2019?." https://blog.techmagic.co/native-vs-hybrid-apps/, 2019. Accessed on 04-20-2021.

[6] Facebook Inc., "React Native." https://facebook.github.io/react-native/. Accessed on 04-20-2021.

[7] B. S. Max Lynch and A. Bradle, "Ionic Framework." https://ionicframework.com/. Accessed on 04-20-2021.

[8] V. Kharlampidi, "Framework7." https://framework7.io/. Accessed on 04-20-2021.

[9] Google, "Flutter." https://flutter.dev/. Accessed on 04-20-2021.

[10] Monaca, Inc., "Onsen UI." https://onsen.io/. Accessed on 04-20-2021.

[11] Progress Software, "NativeScript: Create Native iOS and Android Apps with JavaScript." https://www.nativescript.org/. Accessed on 04-20-2021.

[12] Microsoft Corp., "Visual Studio Tools for Xamarin." https://visualstudio.microsoft.com/xamarin/. Accessed on 04-20-2021.

[13] M. Casimirri, "Mobile Angular UI." https://github.com/mcasimir/mobile-angular-ui. Accessed on 04-20-2021.

[14] Ionic, "Developer Survey 2020." https://ionicframework.com/survey/2020#trends. Accessed on 04-20-2021.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proc. of the 18th ACM Conf. on Comp. and Comm. Sec. (CCS '11)*, pp. 627–638, 2011.

[16] M. Shehab and A. AlJarrah, "Reducing attack surface on cordova-based hybrid mobile apps," in *Proc. of the 2nd Intl. Workshop on Mobile Development Lifecycle (MobileDeli)*, pp. 1–8, 2014.

[17] J. Mao, H. Ma, Y. Chen, Y. Jia, and Z. Liang, "Automatic permission inference for hybrid mobile apps," *Journal of High Speed Networks*, pp. 55–64, 2016.

[18] Integral Ad Science, Inc., "Effectively influence consumers everywhere." https://integralads.com/, 2016. Accessed on 04-20-2021.

[19] P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan, "Between worlds: Securing mixed JavaScript/ActionScript multi-party web content," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, pp. 443–457, 2015.

[20] T. Tran, R. Pelizzi, and R. Sekar, "JaTE: Transparent and Efficient JavaScript Confinement," in *Proc. of the 31st Annual Comp. Sec. Applications Conf.*, ACSAC 2015, pp. 151–160, 2015.

[21] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proc. of the 28th Annual Comp. Sec. Applications Conf. (ACSAC)*, pp. 1–10, 2012.

[22] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, "AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements," in *Proc. of USENIX Sec.'10*, pp. 24–41, 2010.

[23] M. T. Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan, "SafeScript: JavaScript Transformation for Policy Enforcement," in *Proc. of the 18th Nordic Conf. on Secure IT Systems (NordSec 2013)*, pp. 67–83, 2013.

[24] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser," in *2010 IEEE Symp. on Sec. and Privacy*, pp. 481–496, 2010.

[25] P. H. Phung and L. Desmet, "A two-tier sandbox architecture for untrusted JavaScript," in *Proc. of the Workshop on JavaScript Tools (JSTools)*, pp. 1–10, 2012.

[26] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," in *Proc. of the 4th Intl. Symp. on Information, Comp., and Comm. Sec. (ASIACCS)*, pp. 47–60, 2009.

[27] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen, "WebJail: Least-privilege Integration of Third-party Components in Web Mashups," in *Proc. of the 27th Annual Comp. Sec. Applications Conf.*, ACSAC '11, pp. 307–316, 2011.

[28] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices," 2019.

[29] K. Nakhaei, E. Ansari, and F. Ansari, "JSSignature: Eliminating Third-Party-Hosted JavaScript Infection Threats Using Digital Signatures," *arXiv preprint arXiv:1812.03939*, 2018.

[30] A. L. S. Pupo, J. Nicolay, and E. G. Boix, "GUARDIA: Specification and Enforcement of Javascript Security Policies without VM Modifications," in *Proc. of the 15th Intl. Conf. on Managed Languages & Runtimes (ManLang '18)*, 2018.

[31] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices," in *Proc. of the 2019 ACM Asia Conf. on Comp. and Comm. Sec. (Asia CCS '19)*, pp. 391–402, 2019.

[32] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on Android," in *Proc. of the 2016 ACM SIGSAC Conf. on Comp. and Comm. Sec. (CCS)*, pp. 104–115, 2016.

[33] X. Jin, L. Wang, T. Luo, and W. Du, "Fine-Grained Access Control for HTML5-Based Mobile Applications in Android," in *Proc. of the 16th Intl. Conf. on Information Sec. - Volume 7807*, pp. 309–318, 2015.

[34] S. Pooryousef and M. Amini, "Fine-Grained Access Control for Hybrid Mobile Applications in Android Using Restricted Paths," in *Proc. of the 13th Intl. ISC Conf. on Information Sec. and Cryptology (ISCISC)*, pp. 85–90, 2016.

[35] A. AlJarrah and M. Shehab, "The demon is in the configuration: Revisiting hybrid mobile apps configuration model," in *Proc. of the 12th Intl. Conf. on Availability, Reliability and Sec. (ARES '17)*, pp. 57:1–57:10, 2017.

[36] A. AlJarrah and M. Shehab, "Closer look at mobile hybrid apps configurations: Statistics and Implications," in *Advances in Information and Communication*, pp. 1016–1037, 2019.

[37] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Proc. of the 21st Annual Network and Distributed System Sec. Symp. (NDSS)*, pp. 1–15, 2014.

[38] N. Kudo, T. Yamauchi, and T. H. Austin, "Access control mechanism to mitigate cordova plugin attacks in hybrid applications," *JIP*, pp. 396–405, 2018.

[39] M. Georgiev, S. Jana, and V. Shmatikov, "Rethinking security of web-based system applications," in *Proc. of the 24th Intl. Conf. on World Wide Web (WWW)*, pp. 366–376, 2015.

[40] Mozilla Development Network (MDN), "Introduction to the DOM." https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. Accessed on 11-17-2019.

[41] A. Holst, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030(in billions)." https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. Accessed on 04-20-2021.

[42] Zion Market Research, "Iot devices market size & share growing rapidly to surpass usd 158,140 million by 2024." https://www.globenewswire.com/news-release/2018/10/15/1621098/0/en/IoT-Devices-Market-Size-Share-Growing-Rapidly-To-Surpass-USD-158-140-Million-by-2024.html. Accessed on 04-20-2021.

[43] E. Fernandes, J. Jung, and A. Prakash, "Sec. Analysis of Emerging Smart Home Applications," in *2016 IEEE Symp. on Sec. and Privacy (SP)*, pp. 636–654, 2016.

[44] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Sec. Evaluation of Home-Based IoT Deployments," in *2019 IEEE Symp. on Sec. and Privacy (SP)*, 2019.

[45] Z. B. Celik, P. McDaniel, and G. Tan, "SOTERIA: Automated IoT Safety and Sec. Analysis," in *Proc. of the 2018 USENIX Conf. on Usenix Annual Technical Conf. (USENIX ATC '18)*, pp. 147–158, 2018.

[46] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the Mirror: Evaluating IoT Device Sec. Through Mobile Companion Apps," in *Proc. of the 28th USENIX Conf. on Sec. Symp. (SEC'19)*, pp. 1151–1167, 2019.

[47] D. M. Junior, L. Melo, H. Lu, M. d'Amorim, and A. Prakash, "Beware of the App! On the Vulnerability Surface of Smart Devices through their Companion Apps," *CoRR*, 2019.

[48] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Network and Distributed System Sec. Symp. (NDSS)*, 2018.

[49] W. Crumpler, "The Cybersecurity Workforce Gap." https://bit.ly/2IZ5snw. Accessed on 04-20-2021.

[50] $(ISC)^2$, "Cybersecurity Professionals Focus on Developing New Skills as Workforce Gap Widens: $(ISC)^2$ CYBERSECURITY WORKFORCE STUDY," tech. rep. Accessed on 04-20-2021.

[51] P. Loshin, "Mcafee CISO explains why diversity in cybersecurity matters." https://searchsecurity.techtarget.com/feature/McAfee-CISO-explains-why-diversity-in-cybersecurity-matters. Accessed on 04-20-2021.

[52] "The Need for Diversity in Cybersecurity." https://medium.com/diversity-unscripted/the-need-for-diversity-in-cybersecurity-1ec1c14e1770. Accessed on 04-20-2021.

[53] DataUSA, "INFORMATION SECURITY ANALYSTS." https://datausa.io/profile/soc/information-security-analysts. Accessed on 04-20-2021.

[54] K. Walsh, "Engaging in diversity, equity, and inclusion for stronger cybersecurity." https://www.securitymagazine.com/articles/93947-engaging-in-diversity-equity-and-inclusion-for-stronger-cybersecurity. Accessed on 04-20-2021.

[55] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar, "Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications," in *Proc. of 2017 IEEE Sec. and Privacy Workshops (SPW) - Mobile Sec. Technologies (MoST)*, pp. 147–156, 2017.

[56] P. H. Phung, R. S. Reddy, S. Cap, A. Pierce, A. Mohanty, and M. Sridhar, "HybridGuard: A Multi-Party, Fine-Grained Permission and Policy Enforcement Framework for Hybrid Mobile Applications," *Journal of Comp. Sec.*, 2020.

[57] A. Mohanty and M. Sridhar, "HybriDiagnostics: An Automated Vulnerability Assessment Framework for Hybrid Smart Home Companion Apps," *Computers and Security Journal (Elsevier)*, 2021. Submitted. In Review.

[58] A. Monus, "Understanding native app development—what you need to know in 2019." https://raygun.com/blog/native-app-development/. Accessed on 04-20-2021.

[59] Y Media Labs (YML), "Native vs Hybrid Mobile Apps—Here's How To Choose." https://uxplanet.org/native-vs-hybrid-mobile-apps-heres-how-to-choose-192ecbf04da8. Accessed on 04-20-2021.

[60] D. Pania, "Understanding Permissions in the Android World." https://clevertap.com/blog/understanding-android-permissions/. Accessed on 04-20-2021.

[61] G. Inc., "Webview." https://developer.android.com/reference/android/webkit/WebView. Accessed on 04-20-2021.

[62] A. Inc., "WKWebView." https://developer.apple.com/documentation/webkit/wkwebview. Accessed on 04-20-2021.

[63] C. Griffith, "What is Apache Cordova?." https://ionic.io/resources/articles/what-is-apache-cordova. Accessed on 04-16-2021.

[64] Apache Cordova Project, "Platform Support—Core Plugin APIs." https://cordova.apache.org/docs/en/10.x/guide/support/index.html#core-plugin-apis. Accessed on 04-16-2021.

[65] npmjs.com, "cordova-plugin-chrome-apps-proxy." https://www.npmjs.com/package/cordova-plugin-chrome-apps-proxy. Accessed on 04-16-2021.

[66] Apache Cordova Project, "Cordova." https://cordova.apache.org/. Accessed on 04-20-2021.

[67] Andreas Barth, "The Web Origin Concept." https://tools.ietf.org/html/rfc6454. Accessed on 04-20-2021.

[68] M. West and J. Medley, "Content Sec. Policy." https://developers.google.com/web/fundamentals/security/csp/. Accessed on 04-20-2021.

[69] Apache Software Foundation, "Allow List Guide." https://cordova.apache.org/docs/en/10.x/guide/appdev/allowlist/index.html. Accessed on 04-20-2021.

[70] Microsoft, "Cordova whitelist and content sec. policy guide." https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/security/whitelists?view=toolsforcordova-2017. Accessed on 04-20-2021.

[71] A. Cordova, "cordova-plugin-whitelist." https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-whitelist/. Accessed on 04-20-2021.

[72] N. van Ginkel, W. D. Groef, F. Massacci, and F. Piessens, "A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries," *Sec. and Communication Networks*, 2019.

[73] I. Aktug and K. Naliuka, "ConSpec—A formal language for policy specification," *Science of Comp. Programming*, pp. 2–12, 2008.

[74] J. Ligatti, B. Rickey, and N. Saigal, "LoPSiL: A Location-Based Policy-Specification Language," in *Sec. and Privacy in Mobile Information and Communication Systems: First Intl. ICST Conf., MobiSec 2009*, pp. 265–277, 2009.

[75] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky, "AppGuard—Enforcing User Requirements on Android Apps," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 543–548, 2013.

[76] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," pp. 658–673, 2012.

[77] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Presented as part of the 22nd USENIX Sec. Symp. (USENIX Sec. 13)*, pp. 131–146, 2013.

[78] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proc. of the 5th ACM symposium on information, computer and communications security*, pp. 328–332, 2010.

[79] Y. Imamura, H. Uekawa, Y. Ishihara, M. Sato, and T. Yamauchi, "Web access monitoring mechanism for Android WebView," in *Proc. of the Australasian Comp. Science Week Multiconference*, pp. 1:1–1:8, 2018.

[80] D. Franzen and D. Aspinall, "PhoneWrap-injecting the "How Often" into mobile apps," in *The 1st Intl. Workshop on Innovations in Mobile Privacy and Sec. (IMPS)*, pp. 11–19, 2011.

[81] K. Singh, "Practical context-aware permission control for hybrid mobile applications," in *Proc. of the 16th Intl. Workshop on Recent Advances in Intrusion Detection (RAID)*, pp. 307–327, 2013.

[82] A. AlJarrah and M. Shehab, "CordovaConfig: A tool for mobile hybrid apps' configuration," in *Proc. of the 17th Intl. Conf. on Mobile and Ubiquitous Multimedia (MUM 2018)*, pp. 161–170, 2018.

[83] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications," in *2018 IEEE Symp. on Sec. and Privacy (SP)*, pp. 742–755, 2018.

[84] Y.-L. Chen, H.-M. Lee, A. B. Jeng, and T.-E. Wei, "DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps," in *Proc. of the 14th Trust, Sec. and Privacy in Computing and Comm. (TRUSTCOM)*, pp. 1014–1021, 2015.

[85] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, and Y. Jiang, "Detection and prevention of code injection attacks on HTML5-based apps," in *Proc. of the 3rd Intl. Conf. on Advanced Cloud and Big Data (CBD*, pp. 254–261, 2015.

[86] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *Journal of Systems and Software*, pp. 67–77, 2018.

[87] J. Mao, R. Wang, Y. Chen, and Y. Jia, "Detecting injected behaviors in html5-based android applications," *Journal of High Speed Networks*, pp. 15–34, 2016.

[88] P. T. Lau, "Scan code injection flaws in HTML5-based mobile applications," in *2018 IEEE Intl. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 81–88, 2018.

[89] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "BridgeTaint: A bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications," *IEEE Transactions on Information Forensics and Sec.*, pp. 677–692, 2019.

[90] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A large-scale study of mobile web app sec.," in *Proc. of the Mobile Sec. Technologies Workshop (MoST)*, 2015.

[91] M. Willocx, J. Vossaert, and V. Naessens, "Sec. analysis of cordova applications in Google Play," in *Proc. of the 12th Intl. Conf. on Availability, Reliability and Sec. (ARES '17)*, pp. 46:1–46:7, 2017.

[92] M. Ali and A. Mesbah, "Mining and characterizing hybrid apps," in *Proc. of the Intl. Workshop on App Market Analytics (WAMA 2016)*, pp. 50–56, 2016.

[93] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and scalably vetting javascript bridge in android hybrid apps," in *RAID*, 2017.

[94] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for android hybrid applications," in *Proc. of the 31st IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pp. 250–261, 2016.

[95] S. Pouryousef, M. Rezaiee, and A. Chizari, "Let me join two worlds! analyzing the integration of web and native technologies in hybrid mobile apps," in *2018 17th IEEE Intl. Conf. On Trust, Sec. And Privacy In Computing And Comm./ 12th IEEE Intl. Conf. On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 1814–1819, 2018.

[96] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, "A survey and taxonomy of core concepts and research challenges in cross-platform mobile development," *ACM Computing Surveys (CSUR)*, pp. 108:1–108:34, 2018.

[97] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel, "IotSan: Fortifying the Safety of IoT Systems," in *Proc. of the 14th Intl. Conf. on Emerging Networking EXperiments and Technologies (CoNEXT '18)*, pp. 191–203, 2018.

[98] T. Denning, A. Lerner, A. Shostack, and T. Kohno, "Control-Alt-Hack: The Design and Evaluation of a Card Game for Comp. Sec. Awareness and Education," in *Proc. of the 2013 ACM SIGSAC Conf. on Comp. & Comm. Sec.*, CCS '13, (New York, NY, USA), pp. 915–928, Association for Computing Machinery, 2013.

[99] Z. C. Schreuders and E. Butterfield, "Gamification for teaching and learning comp. sec. in higher education," in *2016 USENIX Workshop on Advances in Sec. Education (ASE 16)*, (Austin, TX), USENIX Association, Aug. 2016.

[100] Trend Micro: The fugle company, "Targeted attack: The game." http://targetedattacks.trendmicro.com/. Accessed on 04-20-2021.

[101] Matt Trobbiani, "Hacknet Labyrinths." https://store.steampowered.com/app/521840/Hacknet__Labyrinths/. Accessed on 04-20-2021.

[102] C. Li and R. Kulkarni, "Survey of cybersecurity education through gamification," in *2016 ASEE Annual Conf. & Exposition*, no. 10.18260/p.25981, (New Orleans, Louisiana), ASEE Conferences, June 2016. https://peer.asee.org/25981.

[103] M. R. Asghar and A. Luxton-Reilly, "Teaching cyber sec. using competitive software obfuscation and reverse engineering activities," in *Proc. of the 49th ACM Technical Symp. on Comp. Science Education*, SIGCSE '18, (New York, NY, USA), pp. 179–184, Association for Computing Machinery, 2018.

[104] P. Buckley and E. Doyle, "Gamification and student motivation," *Interactive Learning Environments*, vol. 24, no. 6, pp. 1162–1175, 2016.

[105] OverTheWire (community), "Wargames." http://overthewire.org/wargames/. Accessed on 04-20-2021.

[106] S. Watson and H. R. Lipford, "Motivating students beyond course requirements with a serious game," in *Proc. of the 50th ACM Technical Symp. on Comp. Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, (New York, NY, USA), pp. 211–217, Association for Computing Machinery, 2019.

[107] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.

[108] C. Crawford, *Chris Crawford on Game Design*. USA: New Riders Publishing, 2003.

[109] M. Mateas and P. Sengers, "Narrative Intelligence," in *AAAI Fall Symp.*, AAAI Press, 1998.

[110] J. Juul, *Half-real: Video games between real rules and fictional worlds*. MIT press, 2011.

[111] M. Mateas and A. Stern, "Fa{ç}ade: An experiment in building a fully-realized interactive drama," in *Game Developers Conference, Game Design track*, Citeseer, 2003.

[112] D. R. Krathwohl, "A revision of bloom's taxonomy: An overview," *Theory Into Practice*, vol. 41, no. 4, pp. 212–218, 2002.

[113] C. C. Bonwell and J. A. Eison, *Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports.* ERIC, 1991.

[114] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth, "Active learning increases student performance in science, engineering, and mathematics," *Proc. of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410–8415, 2014.

[115] C. Meyers and T. B. Jones, *Promoting Active Learning. Strategies for the College Classroom.* ERIC, 1993.

[116] C. Latulipe, N. B. Long, and C. E. Seminario, "Structuring Flipped Classes with Lightweight Teams and Gamification," in *Proc. of the 46th ACM Technical Symp. on Comp. Science Education*, SIGCSE '15, (New York, NY, USA), pp. 392–397, ACM, 2015.

[117] S. MacNeil, C. Latulipe, B. Long, and A. Yadav, "Exploring Lightweight Teams in a Distributed Learning Environment," in *Proc. of the 47th ACM Technical Symp. on Computing Science Education*, SIGCSE '16, (New York, NY, USA), pp. 193–198, ACM, 2016.

[118] J. Bergmann and A. Sams, *Flip your classroom: Reach every student in every class every day*. Intl. society for technology in education, 2012.

[119] J. L. Bishop and M. A. Verleger, "The flipped classroom: A survey of the research," in *ASEE National Conf. Proc., Atlanta, GA*, vol. 30, pp. 1–18, 2013.

[120] J. Bergmann and A. Sams, *Flipped Learning: Gateway to Student Engagement*. Intl. Society for Technology in Education, 2014.

[121] M. L. Maher, C. Latulipe, H. Lipford, and A. Rorrer, "Flipped Classroom Strategies for CS Education," in *Proc. of the 46th ACM Technical Symp. on Comp. Science Education*, SIGCSE '15, pp. 218–223, 2015.

[122] R. S. Moog, J. N. Spencer, and A. R. Straumanis, "Process-oriented guided inquiry learning: Pogil and the pogil project," *Metropolitan Universities*, vol. 17, no. 4, pp. 41–52, 2006.

[123] R. Moog, *Process oriented guided inquiry learning*. Washington University Libraries, 2014.

[124] "Process oriented guided inquiry learning." https://pogil.org/. Accessed on 04-20-2021.

[125] H. H. Hu and C. Kussmaul, "Promoting student-centered learning with pogil," in *Proc. of the 43rd ACM Technical Symp. on Comp. Science Education*, SIGCSE '12, pp. 579–580, 2012.

[126] "Process Oriented Guided Inquiry Learning." http://cspogil.org/Home. Accessed on 04-20-2021.

[127] C. Kussmaul, "Process oriented guided inquiry learning (pogil) for computer science," in *SIGCSE*, 2012.

[128] H. H. Hu and T. D. Shepherd, "Teaching CS 1 with POGIL activities and roles," in *Proc. of the 45th ACM technical symposium on Comp. science education*, pp. 127–132, ACM, 2014.

[129] A. Brucker, "Hybrid Mobile Apps: From Security Challenges to Secure Development." https://www.brighttalk.com/webcast/288/271705/hybrid-mobile-apps-from-security-challenges-to-secure-development. Accessed: 2019-5-7.

[130] Tactical Network Solutions, "IoT Firmware Exploitation." https://www.tacnetsol.com/store/aRyibNKX. Accessed on 04-20-2021.

[131] Attify, "Offensive IoT Exploitation." https://www.attify.com/iot-security-exploitation-training. Accessed on 04-20-2021.

[132] Tonex, "Iot sec. training." https://www.tonex.com/training-courses/iot-security-training-iot-security-awareness/. Accessed on 04-20-2021.

[133] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker.," in *Proc. of the USENIX Annual Technical Conf.*, pp. 309–318, 2012.

[134] M. Zalewski, "American fuzzy lop: a security-oriented fuzzer," 2010.

[135] Udemy, "Fundamentals of iot sec." https://www.udemy.com/fundamentals-of-iot-security. Accessed on 04-20-2021.

[136] edx, "Cybersecurity and Privacy in the IoT." https://www.edx.org/course/cybersecurity-and-privacy-in-the-iot. Accessed on 04-20-2021.

[137] Brian Russel and Sunil Gupta, "Securing IoT: From Security to Practical Pentesting on IoT." https://www.udemy.com/course/securing-iot-from-security-to-practical-pentesting-on-iot/. Accessed on 04-20-2021.

[138] edX-Curtin University, "Cybersecurity and Privacy in the IoT." https://www.edx.org/course/cybersecurity-and-privacy-in-the-iot. Accessed on 04-20-2021.

[139] T. Chothia and J. de Ruiter, "Learning From Others' Mistakes: Penetration Testing IoT Devices in the Classroom," in *2016 USENIX Workshop on Advances in Sec. Education (ASE 16)*, 2016.

[140] F. B. Schneider, "Enforceable Sec. Policies," *ACM Transactions on Information and System Sec. (TISSEC)*, pp. 30–50, 2000.

[141] Apache Cordova, "Security Guide." https://cordova.apache.org/docs/en/10.x/guide/appdev/security/index.html. Accessed on 04-20-2021.

[142] S. Butner, "How much in advertising revenue can a mobile app generate?." http://smallbusiness.chron.com/much-advertising-revenue-can-mobile-app-generate-76855.html. Accessed on 04-20-2021.

[143] W. Zamora, "Truth in malvertising: How to beat bad ads." http://bit.ly/how-to-beat-bad-ads. Accessed on 04-20-2021.

[144] M. Rouse, "Malvertisement (malicious advertisement or malvertising)." http://searchsecurity.techtarget.com/definition/malvertisement-malicious-advertisement-or-malvertising. Accessed on 04-20-2021.

[145] A. Hern, "Spotify hit by 'malvertising' in app." https://www.theguardian.com/technology/2016/oct/06/spotify-hit-by-malvertising-in-app. Accessed on 04-20-2021.

[146] J. Kirk, "Massive malvertising campaign hits msn, yahoo." http://www.bankinfosecurity.com/massive-malvertising-campaign-hits-msn-yahoo-a-9583. Accessed on 04-20-2021.

[147] D. Goodin, "Millions exposed to malvertising that hid attack code in banner pixels." http://bit.ly/arstechnica-malvertising. Accessed on 04-20-2021.

[148] OWASP, "Clickjacking." https://www.owasp.org/index.php/Clickjacking. Accessed on 04-20-2021.

[149] Y. Qiu, "Tapjacking: An untapped threat in android." https://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/. Accessed on 04-20-2021.

[150] iBotPeaches, "Apktool." https://ibotpeaches.github.io/Apktool/. Accessed on 04-20-2021.

[151] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly Media, 6th ed., 2011.

[152] B. Alpern and F. B. Schneider, "Defining liveness," tech. rep., 1984.

[153] W3C Schools, "What is JSON?." https://www.w3schools.com/whatis/whatis_json.asp. Accessed on 04-20-2021.

[154] W3C, "Cross-Origin Resource Sharing." https://www.w3.org/TR/cors/. Accessed on 04-20-2021.

[155] Mozilla Development Network (MDN), "Inheritance and the prototype chain." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain. Accessed on 04-20-2021.

[156] J. Magazinius, P. H. Phung, and D. Sands, "Safe wrappers and sane policies for self protecting JavaScript," in *Proc. of the 15th Nordic Conf. in Secure IT Systems (NordSec)*, pp. 239–255, 2010.

[157] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 3, p. 11, 2007.

[158] A. Takyar, "Iot app development: Impact, challenges, and process." https://dzone.com/articles/iot-app-development-impact-challenges-and-process. Accessed on 04-20-2021.

[159] finjanmobile, "Apps are creating mobile security vulnerabilities for iot—how bad is it?." https://www.finjanmobile.com/mobile-security-vulnerabilities-for-iot/. Accessed on 04-20-2021.

[160] W. Ashford, "Iot application vulnerabilities leave devices open to attack." https://www.computerweekly.com/news/252456406/IoT-application-vulnerabilities-leave-devices-open-to-attack. Accessed on 04-20-2021.

[161] S. Tenaglia, "Breaking BHAD: Injecting code into the WeMo app using XSS." https://www.twosixlabs.com/breaking-bhad-injecting-code-into-the-wemo-app-using-xss/, 2016. Accessed on 04-16-2021.

[162] McAfee, "McAfee labs 2016 threats predictions." https://tinyurl.com/y9vqs44h, September 2016. Retrieved 10-1-2016.

[163] toptal.com, "Hire Hybrid App Developers." https://www.toptal.com/hybrid-app. Accessed on 04-16-2021.

[164] goodfirms.co, "Top Hybrid App Development Companies." https://www.goodfirms.co/directory/platform/app-development/hybrid. Accessed on 04-16-2021.

[165] V. Sivaraman, D. Chan, D. Earl, and R. Boreli, "Smart-Phones Attacking Smart-Homes," in *Proc. of the ACM Conf. on Security and Privacy in Wireless and Mobile Networks, (WiSec)*, pp. 195–200, 2016.

[166] Adobe Inc., "Adobe PhoneGap." https://phonegap.com/. Accessed on 04-20-2021.

[167] StackOverflow, "Developer Survey Results 2020." https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved. Accessed on 04-20-2021.

[168] Twinkle, "Best 5 Hybrid App Frameworks that you need to make rocking Apps in 2020." https://www.mobileappdaily.com/best-hybrid-app-frameworks. Accessed on 04-20-2021.

[169] M. Ali and A. Mesbah, "Mining and Characterizing Hybrid Apps," in *Proc. of the Intl. Workshop on App Market Analytics*, pp. 50–56, 2016.

[170] Google, "DexDump." https://android.googlesource.com/platform/dalvik/+/09239e3/dexdump/DexDump.c. Accessed on 04-20-2021.

[171] J. Kaneko, "Understanding React Native Architecture." https://dev.to/goodpic/understanding-react-native-architecture-22hh, 2020. Accessed on 04-16-2021.

[172] React Developer Docs, "Introducing JSX." https://reactjs.org/docs/introducing-jsx.html. Accessed on 04-16-2021.

[173] React Developer Docs, "JSX Prevents Injection Attacks." https://reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks. Accessed on 04-16-2021.

[174] PowerBrick.NET, "PowerBrick Broadband Monitoring Solutions." http://www.powerbrick.net/Shop/Default.aspx?Group=5. Accessed on 04-16-2021.

[175] Nedis, "Nedis Smart Home Products." https://nedis.com/en-us/category/security-safety/smart-home. Accessed on 04-16-2021.

[176] A. Niakanlahiji, B. Chu, and E. Al-Shaer, "PhishMon: A Machine Learning Framework for Detecting Phishing Webpages," in *2018 IEEE International Conf. on Intelligence and Security Informatics (ISI)*, pp. 220–225, 2018.

[177] ssdeep Project, "ssdeep—Fuzzy hashing program." https://ssdeep-project.github.io/ssdeep/index.html. Accessed on 04-16-2021.

[178] S. Larsen, "Why It's Bad to Use 'unsafe-inline' in script-src." https://csper.io/blog/no-more-unsafe-inline. Accessed on 04-20-2021.

[179] William Le, "Reasons Why You Should Never Use eval() in JavaScript." https://www.digitalocean.com/community/tutorials/js-eval, 2019. Accessed on 04-16-2021.

[180] S. Alon, "CSP bypasses, and how developers can build a strict CSP!." https://cspscanner.com/csp-bypasses#why-are-developers-creating-bad-csps. Accessed on 04-16-2021.

[181] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy," in *Proc. of the ACM SIGSAC Conf. on Comp. and Comm. Security (CCS)*, pp. 1376–1387, 2016.

[182] S. Calzavara, A. Rabitti, and M. Bugliesi, "Semantics-Based Analysis of Content Security Policy Deployment," *ACM Transactions on the Web (TWEB)*, 2018.

[183] RapidSec, "CSP bypasses, and how developers can build a strict CSP!." https://cspscanner.com/csp-bypasses. Accessed on 04-16-2021.

[184] Fortify, "HTML5: Misconfigured Content Security Policy." https://vulncat.fortify.com/en/detail?id=desc.dynamic.html.html5_misconfigured_content_security_policy. Accessed on 04-16-2021.

[185] Detectify, "Content Security Policy (CSP) explained including common bypasses." https://blog.detectify.com/2019/07/11/content-security-policy-csp-explained-including-common-bypasses/. Accessed on 04-16-2021.

[186] G. Scalzi, "Content Security Policy: misconfigurations and bypasses." https://blog.compass-security.com/2016/06/content-security-policy-misconfigurations-and-bypasses/. Accessed on 04-16-2021.

[187] Google Open Source Project, "CSP Evaluator Core Library." https://github.com/google/csp-evaluator. Accessed on 04-16-2021.

[188] Micron Security Innovation, "PowerBrick Alarm App." https://play.google.com/store/apps/details?id=com.micronsecurity.pengyanb.powerbrickalarm&hl=en_US&gl=US. Accessed on 04-16-2021.

[189] OWASP, "XSS Filter Evasion Cheat Sheet." https://owasp.org/www-community/xss-filter-evasion-cheatsheet. Accessed on 04-16-2021.

[190] M. Weissbacher, T. Lauinger, and W. Robertson, "Why Is CSP Failing? Trends and Challenges in CSP Adoption," in *Proc. of Intl. Symp. on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 212–233, 2014.

[191] J. Weinberger, A. Barth, and D. Song, "Towards Client-Side HTML Security Policies," in *Proc. of the USENIX Conf. on Hot Topics in Security (HotSec)*, 2011.

[192] i4Home, "i4Home." https://play.google.com/store/apps/details?id=com.i4home.livingpattern&hl=en_US&gl=US. Accessed on 04-16-2021.

[193] MDN Web Docs, "eval()." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval. Accessed on 04-16-2021.

[194] Apache Cordova Docs, "Storage." https://cordova.apache.org/docs/en/latest/cordova/storage/storage.html. Accessed on 04-16-2021.

[195] OWASP, "OWASP Mobile Top 10." https://owasp.org/www-project-mobile-top-10/. Accessed on 04-16-2021.

[196] npmjs.com, "cordova-plugin-secure-storage." https://www.npmjs.com/package/cordova-plugin-secure-storage. Accessed on 04-16-2021.

[197] brodybits on GitHub, "cordova-sqlcipher-adapter." https://github.com/brodybits/cordova-sqlcipher-adapter. Accessed on 04-16-2021.

[198] npmjs.com, "com-intel-security-cordova-plugin." https://www.npmjs.com/package/com-intel-security-cordova-plugin. Accessed on 04-16-2021.

[199] A. Bilisim, "Arnido Smart Home." https://play.google.com/store/apps/details?id=com.armakom.arnidosmarthome&hl=en_US&gl=US. Accessed on 04-16-2021.

[200] Android Developers Documentation, "Intent." https://developer.android.com/reference/android/content/Intent, 2020. Accessed on 04-20-2021.

[201] MITRE, "CVE-2015-1835." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1835. Accessed on 04-16-2021.

[202] MITRE, "CVE-2019-0219." https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-0219. Accessed on 04-16-2021.

[203] MITRE CVE Database, "Apache Cordova CVEs." https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=cordova. Accessed on 04-16-2021.

[204] Apache Cordova Team, "Cordova Blog." https://cordova.apache.org/blog/. Accessed on 04-16-2021.

[205] MITRE, "CVE Database." https://cve.mitre.org/. Accessed on 04-16-2021.

[206] MITRE, "CWE Common Weakness Enumeration." https://cwe.mitre.org/. Accessed on 04-16-2021.

[207] Forum of Incident Response and Security Teams (FIRST), "Common Vulnerability Scoring System SIG." https://www.first.org/cvss/. Accessed on 04-16-2021.

[208] omguardec2, "Smart Home Security." https://play.google.com/store/apps/details?id=com.nedis.smarthomesecurity. Accessed on 04-16-2021.

[209] Multiple Contributors, "Apache Cordova Android (CordovaActivity.java)." https://github.com/apache/cordova-android/blob/master/framework/src/org/apache/cordova/CordovaActivity.java. Accessed on 04-16-2021.

[210] Google, "Activity." https://developer.android.com/reference/android/app/Activity. Accessed on 04-16-2021.

[211] Trendmicro, "Apache vulnerability allows android app modification." https://bit.ly/2NVTQEi. Accessed on 04-16-2021.

[212] Imperva.com, "Phishing attacks." https://www.imperva.com/learn/application-security/phishing-attack-scam/. Accessed on 04-16-2021.

[213] Kaspersky.com, "What is a drive by download: Everything you need to know." https://www.kaspersky.com/resource-center/definitions/drive-by-download. Accessed on 04-16-2021.

[214] F. H. Shezan, S. F. Afroze, and A. Iqbal, "Vulnerability detection in recent Android apps: An empirical study," in *Proc. of International Conf. on Networking, Systems and Security (NSysS)*, pp. 55–63, 2017.

[215] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface," in *Proc. of the Revised Selected Papers of the International Workshop on Security Protocols XXIII* , vol. 9379, (Berlin, Heidelberg), pp. 126–138, Springer-Verlag, 2015.

[216] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications," in *Proc. of the IEEE Symp. on Security and Privacy (SP)*, pp. 742–755, 2018.

[217] Erez Yelon, "Android WebView: Secure Coding Practices." https://www.checkmarx.com/2017/11/16/android-webview-secure-coding-practices/. Accessed on 04-20-2021.

[218] JoÃ£o Morais, "Android WebView: Are Secure Coding Practices Being Followed?." https://securityboulevard.com/2018/12/android-webview-are-secure-coding-practices-being-followed/. Accessed on 04-20-2021.

[219] Lily Hay Newman, "An Android Vulnerability Went Unfixed for Over Five Years." https://www.wired.com/story/android-vulnerability-five-years-fragmentation/. Accessed on 04-20-2021.

[220] Checkmarx, "About Us." https://www.checkmarx.com/about-checkmarx/. Accessed on 04-16-2021.

[221] K. Chivers, "What is a man-in-the-middle attack?." https://us.norton.com/internetsecurity-wifi-what-is-a-man-in-the-middle-attack.html. Accessed on 04-16-2021.

[222] Leviton Manufacturing Co., Inc., "My Leviton." https://play.google.com/store/apps/details?id=com.leviton.home. Accessed on 04-16-2021.

[223] Decorasmartsupport, "Updating firmware in decora smart with homekit technology devices." https://bit.ly/3t4OK83. Accessed on 04-20-2021.

[224] W3Schools, "Html <iframe> tag." https://www.w3schools.com/tags/tag_iframe.asp. Accessed on 04-20-2021.

[225] Apache Cordova, "Security Guide." https://cordova.apache.org/docs/en/latest/guide/appdev/security/. Accessed on 04-20-2021.

[226] Rogers Communication Inc., "Rogers Smart Home Monitoring." https://play.google.com/store/apps/details?id=com.ucontrol.activity. Accessed on 04-16-2021.

[227] Android Developers Documentation, "Meet Google Play's target API level requirement." https://developer.android.com/distribute/best-practices/develop/target-sdk. Accessed on 04-16-2021.

[228] A. Conway, "How Monthly Android Security Patch Updates Work." https://www.xda-developers.com/how-android-security-patch-updates-work/. Accessed on 04-16-2021.

[229] Android Police Blog, "Android security update tracker, March 2021: Rankings for popular smartphones." https://www.xda-developers.com/how-android-security-patch-updates-work/. Accessed on 04-16-2021.

[230] BlueJeans Support, "Android mobile app: End of support for os versions 6 and 7." https://support.bluejeans.com/s/article/android-ver6-7-end-of-support, 2020. Accessed on 04-16-2021.

[231] B. Russel, "All app updates submitted to google play are now required to target android 10 and above." https://www.xda-developers.com/all-apps-google-play-required-target-android-10-api-level-29/. Accessed on 04-16-2021.

[232] Android Developers Documentation, "App Manifest Overview." https://developer.android.com/guide/topics/manifest/manifest-intro, 2020. Accessed on 04-20-2021.

[233] Android Developers Documentation, "<application>." https://developer.android.com/guide/topics/manifest/application-element, 2020. Accessed on 04-20-2021.

[234] Android Developers Documentation, "<uses-permission>." https://developer.android.com/guide/topics/manifest/uses-permission-element, 2020. Accessed on 04-20-2021.

[235] Android Developers Documentation, "<uses-sdk>." https://developer.android.com/guide/topics/manifest/uses-sdk-element, 2020. Accessed on 04-20-2021.

[236] iConservo, "Blossom—Smart Watering." https://play.google.com/store/apps/details?id=com.iconservo.blossom. Accessed on 04-16-2021.

[237] OWASP, "Buffer Overflows." https://www.owasp.org/index.php/Buffer_Overflow. Accessed on 04-20-2021.

[238] laginimaineb, "War of the Worlds—Hijacking the Linux Kernel from QSEE." https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html. Accessed on 04-16-2021.

[239] T. Zahavi-Brunner, "CVE-2017-13253: Buffer overflow in multiple Android DRM services." https://blog.zimperium.com/cve-2017-13253-buffer-overflow-multiple-android-drm-services/. Accessed on 04-16-2021.

[240] Oracle Docs, "Debug a hybrid mobile application on android." https://docs.oracle.com/en/middleware/developer-tools/jet/tutorials/jetmd/index.html. Accessed on 04-16-2021.

[241] MDN Web Docs, "Function." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function. Accessed on 04-16-2021.

[242] Open Source Project on GitHub, "validator.js." https://github.com/validatorjs/validator.js. Accessed on 04-16-2021.

[243] MDN Web Docs, "HTML Sanitizer API." https://developer.mozilla.org/en-US/docs/Web/API/HTML_Sanitizer_API#sanitizer_api_concepts_and_usage. Accessed on 04-16-2021.

[244] Google Open Source Project, "Closure Library." https://github.com/google/closure-library. Accessed on 04-16-2021.

[245] M. Docs, "Cordova platform security features." https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/security/best-practices?view=toolsforcordova-2017. Accessed on 04-16-2021.

[246] E. Yalon, "Android webview: Secure coding practices." https://dzone.com/articles/android-webview-secure-coding-practices. Accessed on 04-20-2021.

[247] R. Kuiper, "How to create a solid and secure Content Security Policy." https://www.uriports.com/blog/creating-a-content-security-policy-csp/, 2020. Accessed on 04-16-2021.

[248] MDN Web Docs, "Form validation." https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation. Accessed on 04-16-2021.

[249] PortSwigger, "Using Burp to Bypass Client Side JavaScript Validation." https://portswigger.net/support/using-burp-to-bypass-client-side-javascript-validation. Accessed on 04-16-2021.

[250] Open Source Project on GitHub, "Yup." https://github.com/jquense/yup. Accessed on 04-16-2021.

[251] Translatehouse.org, "Escaping." http://docs.translatehouse.org/projects/localization-guide/en/latest/guide/translation/escaping.html. Accessed on 04-16-2021.

[252] A. Hamila, "What sanitize mean ? and why sanitize in code/data ?." https://medium.com/@abderrahman.hamila/what-sanitize-mean-and-why-sanitize-in-code-data-5c68c9f76164. Accessed on 04-16-2021.

[253] Open Source Project, "sanitize-html." https://www.npmjs.com/package/sanitize-html. Accessed on 04-16-2021.

[254] Apache on GitHub, "Cordova-Android." https://github.com/apache/cordova-android. Accessed on 04-16-2021.

[255] Josh Fruhlinger, "What is SSL, TLS? And how this encryption protocol works." https://www.csoonline.com/article/3246212/what-is-ssl-tls-and-how-this-encryption-protocol-works.html. Accessed on 04-16-2021.

[256] Google, "`WebViewClient`." https://developer.android.com/reference/android/webkit/WebViewClient. Accessed on 04-16-2021.

[257] F. B. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, pp. 30–50, 2000.

[258] Security Today, "The IoT Rundown For 2020: Stats, Risks, and Solutions." https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=2. Accessed on 04-20-2021.

[259] Nokia, "Nokia Threat Intelligence Report warns of rising cyberattacks on internet-connected devices." https://nokia.ly/3azsLiV. Accessed on 04-20-2021.

[260] Z. Fitz-Walter, "What is gamification?." https://www.gamify.com/what-is-gamification. Accessed on 04-20-2021.

[261] TeachThought Staff, "12 examples of gamification in the classroom." https://www.teachthought.com/the-future-of-learning/12-examples-of-gamification-in-the-classroom/. Accessed on 04-20-2021.

[262] GiantBomb.com, "Experience points." https://www.giantbomb.com/experience-points/3015-39/. Accessed on 04-20-2021.

[263] C. Heffner, "Binwalk: Firmware analysis tool," 2010.

[264] Pallets, "Flask—web development, one drop at a time." https://flask.palletsprojects.com/en/1.1.x/. Accessed on 04-20-2021.

[265] MongoDB, Inc., "MongoDB—The database for modern applications." https://www.mongodb.com/. Accessed on 04-20-2021.

[266] Distributed Management Task Force (DMTF), "Open Virtualization Format." https://www.dmtf.org/standards/ovf. Accessed on 01-14-2021.

[267] M. Rouse, "VMware snapshot." https://searchvmware.techtarget.com/definition/VMware-snapshot. Accessed on 04-20-2021.

[268] wingkwong on Github, "react-quiz-component." https://github.com/wingkwong/react-quiz-component. Accessed on 04-20-2021.