Analyzing Access Control logic in the Android Automotive Framework

by

Jumana

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Android Automotive Operating System (AAOS) is a specialized version of the Android OS designed specifically for in-vehicle hardware. Prominent car manufacturers, including Honda, General Motors (GM), Volvo, and Ford have already adopted it, with Porsche planning to follow soon [62]. Despite its popularity, little has been done to evaluate the security of AAOS integration, particularly at the framework layer where access control vulnerabilities are likely to arise. To bridge the gap, we perform the first security evaluation of automotive APIs in AAOS. Our study is enabled by *AutoAcRaptor*, an automated tool that identifies automotive-specific entry points, generates their access control specifications, and analyzes them for potential security risks. *AutoAcRaptor* leverages static analysis and NLP to perform a three-staged analysis pipeline: 1) Convergence Analysis, 2) Similarity Analysis, and 3) Cross-Image Analysis. Our evaluation demonstrates that the tool is able to efficiently focus the security analysis on auto-specific functionality and pinpoint automotive APIs with likely anomalous access control.

Acknowledgements

I would like to extend my deepest gratitude to the individuals who have supported me throughout the journey of completing this thesis.

First and foremost, I would like to thank my supervisor, Prof. Yousra Aafer, for her invaluable guidance, encouragement, and support throughout this research. Your expertise and insights have been instrumental in shaping this work.

I would also like to express my sincere appreciation to my thesis committee members, Prof. Diogo Barradas and Prof. Meng Xu, for their valuable feedback and suggestions, which greatly improved the quality of this thesis.

I am also grateful to my fellow researcher, Parjanya Vyas, for his camaraderie and collaboration. Your support and stimulating discussions made this journey enjoyable and enlightening.

My friends and family, thank you for your unwavering support and understanding throughout this process. Your encouragement and belief in me have been a source of strength.

Dedication

I dedicate this thesis to my parents and my elder brother, whose constant love and support have been my inspiration.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android Automotive OS (AAOS), launched in 2017 and first implemented in production vehicles in 2020 [9], has rapidly gained prominence in the automotive industry as a fully integrated operating system designed specifically for vehicles. Unlike Android Auto, which simply mirrors smartphone content on infotainment screens, AAOS is embedded in the vehicle's system. It manages in-vehicle functions such as entertainment, navigation, and vehicle system control, and offers automakers the flexibility to tailor them for their needs. Leading automakers, including Volvo, Polestar, and Stellantis, have already integrated AAOS into their vehicles, with others such as Ford and Porsche expected to follow this [62].

Despite its widespread adoption, the security of AAOS, particularly in the framework layer where most changes have taken place to customize AOSP for AAOS integration, has been insufficiently investigated, if at all. As shown by prior literature [48], similar changes to the framework for other purposes (e.g., multi-user framework integration or generic vendor functionality customization) are known to introduce framework-level security vulnerabilities, particularly those related to access control anomalies. In automotive systems, such anomalies are likely to have a more pronounced impact given the diverse and critical resources they manage (e.g., driving control, telematics, etc).

To bridge the gap, we conduct the first systematic investigation of AAOS access control implementation at the framework layer. Our investigation aims to unveil potential anomalies, introduced at new car-specific entry points (i.e., framework code such as APIs and broadcast receivers reachable to apps on the vehicles) or at existing modified entry points (original app-reachable AOSP code modified by vendors).

To this end, we put forward a new automated analysis pipeline for AAOS (which is an enhanced version of previous traditional approaches for Android OS), *AutoAcRaptor* that (1) builds a security specification for (new and modified) AAOS entry points and underlying resources, and (2) leverages the specification to assess the correctness and validity of implemented access control. Specifically, given an automotive framework, *AutoAcRaptor* begins by locating auto-specific code entry points using a few rules (e.g., APIs defined in *Car* system services, APIs checking for automotive features via `PackageManager. hasSystemFeature(FEATURE_AUTOMOTIVE)`, and APIs using variable names with car-related keywords such as *Automotive*, and *car*). For each identified entry point, *AutoAcRaptor* then generates its access control and feature specification through static analysis. To evaluate the specification, it relies on a synergy of approaches (convergence analysis, similarity analysis, cross-ROM analysis) to group functionally-similar entry points and uses that as an oracle to check for anomalies. The key intuition is that two entry points leading to the same resource should enforce similar access control; otherwise, the weakly protected entry is a likely anomaly.

We run *AutoAcRaptor* on ten AAOS ROMs. Among them, two correspond to the latest AAOS AOSP ROMs available on AOSP repositories (versions 13 and 14), and eight are vendor ROMs, customized by four automakers: General Motors (GM), Honda, Volvo and Polestar.

Results and Insights. On average, *AutoAcRaptor* identifies 155 traditional (traditional system services are standard framework services shared with non-AAOS Android) and 34 car system service classes per ROM. Of the entry points, on average, 1.6% are traditional automotive entry points, while 20.7% are car-specific entry points. Notably, the proportion of automotive (car-specific) APIs has increased in newer ROM versions. Additionally, the tool detects that about 9.6% of entry points are customized, with significant modifications from GM, Volvo and Polestar. 12.9% of these customized entry points being newly introduced and about 87.1% existing AOSP entry points modified by the vendors. Besides, *AutoAcRaptor* reveals that, on average, 41.9% of the entry points incorporated at least one access control. Among these, 28.0% were car-specific entry points, 3.8% were traditional automotive entry points, 0.32% were custom entry points (i.e., newly defined by the vendors) and 7.5% were AOSP entry points (i.e., modified by the vendors). Furthermore, *AutoAcRaptor* reports that on average 144 entry point pairs converge and about 55 entry point pairs have naming similarity, that is, pairs providing (partial or full) similar functionality. Of these, about 57% of converging APIs and about 12% of similar APIs enforce

different access controls, respectively, indicating potential anomalies. *AutoAcRaptor* significantly reduces the effort needed to perform this systematic security evaluation of Android AAOS compared to the manual analysis that would be required without it. It analyzes each ROM in approximately 17 minutes, making it manageable for developers to inspect reported anomalies.

Contributions. We make the following contributions:

- We conduct the first systematic security investigation of AAOS frameworks, with regards to access control implementations.
- We design and implement *AutoAcRaptor* a pipeline that combines static analysis and light NLP to generate and evaluate access control specification of AAOS-specific entry points at the framework layer.
- We evaluate our tool on ten AAOS ROMs, quantifying the extent of AAOS customizations, since they generally introduce access control anomalies. Our analysis shows that customizations in automotive entry points, indeed, are particularly prone to these anomalies.

Chapter 2

Background

In this section, we provide necessary background on the AAOS.

2.1 Android Automotive

AAOS is an Android OS version tailored for in-vehicle hardware, providing a platform for automotive infotainment systems. Unlike Android Auto, which requires a smartphone and projects its interface to the car, AAOS runs natively on the vehicle's hardware and connects to in-car networks like the CAN bus [38]. Android Auto provides a driver-optimized app experience via a phone, while AAOS allows users to install apps directly onto the car's system [1]. AAOS leverages Android's features, and security logic while adding automotive-specific enhancements. Google Automotive Services (GAS) are often integrated into AAOS at the discretion of OEMs to enhance the user experience by providing optional services such as Maps, Assistant, and Play Store.

2.2 Evolution of Android to Android Automotive

The Android OS has evolved towards implementing AAOS, by introducing three key components: car-specific managers, car services, and the Vehicle Hardware Abstraction Layer (VHAL). Figure 2.1(a) illustrates the added components in the Android software stack (in green) to tailor Android OS to AAOS. They enable handling automotive-specific functionalities such as vehicle diagnostics, sensor data processing, and communication with various

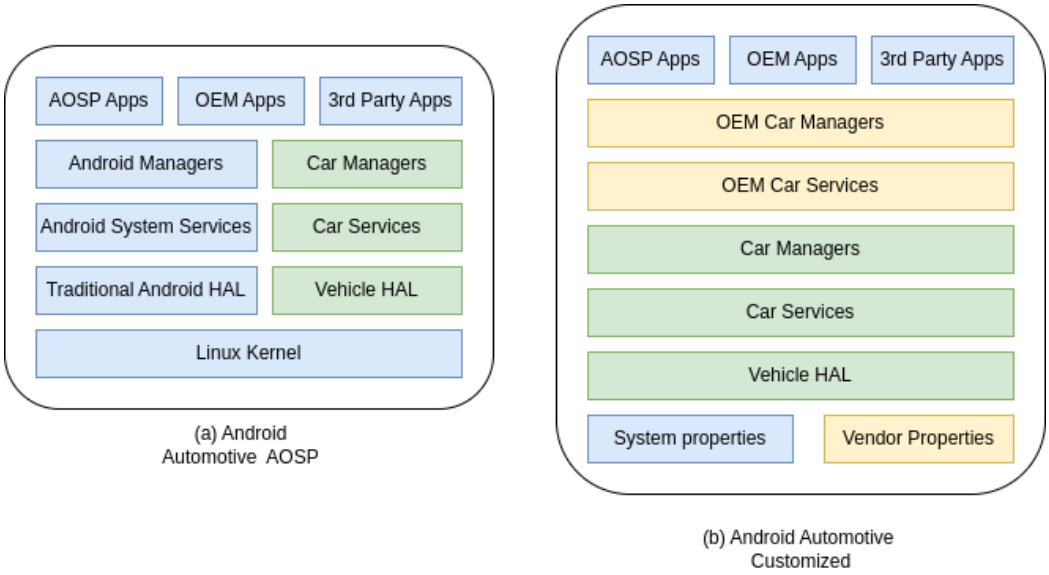

Figure 2.1: The evolution of the Android framework to support the car/automotive functionalities to how the OEMs can customize.

in-car systems (e.g., infotainment, climate control) on top of Android's general-purpose capabilities.

As illustrated in Figure 2.1(b), this evolution further continues when AAOS reaches automakers, such GM or Honda, which revise existing AAOS car managers and services, add in new ones, and integrate new supported vehicle properties to tailor the vehicles for their own functionality and hardware (in yellow).

2.3 Communication Flow in AAOS

Automotive apps can communicate with car-specific system services through the `Car` API using Binder IPC. The `CarService`, which encapsulates all car-specific services, is started by the system server during boot-up. It communicates with Android's standard system services and acts as a bridge to the vehicle's Hardware Abstraction Layer (HAL). The vehicle HAL interfaces with the vehicle's electronic control units (ECUs) over the CAN bus [41], a specialized in-vehicle area network. The ECUs send data, like vehicle speed, to

the vehicle HAL, which stores and presents this information as vehicle properties.

2.4 Characterizing Automotive-Specific Additions/Modifications

2.4.1 New Car Services

AAOS integrates a new library, *car-lib* [27], which defines car-specific system services and their corresponding APIs. Similar to traditional Android APIs, car-specific APIs provide interfaces for vendor and third-party car apps to interact with vehicle-specific framework functionality. Next, we discuss the components of the *car-lib* in detail:

Car API. AAOS introduces a unified *Car API* that is accessible by creating an instance of the *Car* class. Apps can use this instance to access various car-specific services and invoke their exposed API methods. Similar to traditional Android system services (i.e., those common to non-AAOS Android), the AAOS sdk includes *manager* classes for each car-specific service. The managers define public sdk methods that internally invoke car-specific system service APIs (i.e., the managers are client-side wrappers around the system service APIs). Invoking the APIs triggers a binder IPC since they reside on the Android server-side. Figure 2.3 shows an app that uses the Car API to first access an instance of `CarPropertyManager` (Line 4) and then calls two APIs `getProperty` (Line 5) and `getPropertyList` (Line 6).

CarService Design. Android registers and manages car-specific services (e.g., `CarPowerManagementService`, `CarPropertyService`, `CarAudioService`) differently from traditional system services (e.g., `PackageManagerService`, `WindowManagerService`). While the latter are centrally registered in the *ServiceManager* [11], car-specific services are managed by a special system service called *CarService* (registered in the *ServiceManager* and implemented by `ICarImpl` shown in Figure 2.2). *CarService* maintains a comprehensive list of binder car-specific services and can be queried by apps to provide an instance of a car service on demand. Specifically, it exposes a method `getCarService` that accepts a name of car-specific service and returns its corresponding binder instance.

```
// ICarImpl class
public class ICarImpl extends ICar.Stub {
    ...
    // Storing all the car services in the order of their init.
    private final CarSystemService[] mAllServicesInInitOrder;
    ...
    private ICarImpl(Builder builder) {
        TimingsTraceLog t = new TimingsTraceLog(CAR_SERVICE_INIT_TIMING_TAG, TraceHelper.
            TRACE_TAG_CAR_SERVICE, CAR_SERVICE_INIT_TIMING_MIN_DURATION_MS);
        t.traceBegin("ICarImpl.constructor");
        ....
         // Currently there are ~36 services, hence using 40 as the initial capacity.
        List<CarSystemService> allServices = new ArrayList<>(40);
        ....
        // registration of CarOemProxyService
        mCarOemService = constructWithTrace(t, CarOemProxyService.class, () -> new
            CarOemProxyService(mContext), allServices);
         ....
        // registration of CarPropertyService
        mCarPropertyService = constructWithTrace(t, CarPropertyService.class, () -> new
            CarPropertyService.Builder()
        .setContext(mContext)
        .setPropertyHalService(mHal.getPropertyHal())
        .build(), allServices);
        ...
        // registration of CarUserService
         CarLocalServices.addService(CarUserService.class, carUserService);
        ...
         mAllServicesInInitOrder = allServices.toArray(new CarSystemService[allServices.
             size()]);
        ...
    }
    // implementation of constructWithTrace
    private static <T extends CarSystemService> constructWithTrace(TimingsTraceLog t,
        Class<T> cls, Callable<T> callable, List<CarSystemService> allServices) {
        t.traceBegin(cls.getSimpleName());
        T constructed;
        try {
            constructed = callable.call();
            CarLocalServices.addService(cls, constructed);
        } catch (Exception e) {
            throw new RuntimeException("Crash while constructing:" + cls.getSimpleName(),
                e);
        } finally {
            t.traceEnd();
        }
        allServices.add(constructed);
        return constructed;
    }
}
```

Figure 2.2: Simplified Implementation of the Registration of Car System Services.

Specific Car Services' Registration Design. As shown in Figure 2.2, *CarService* registers car-specific services in the method `constructWithTrace` (Line 30) as follows: (1) it creates a new binder instance of a car-specific service (Line 34), (2) adds it to the registry of local services using `CarLocalServices.addService` (Line 35), and (3) adds it to the list of active services (Line 41). Line 15 - Line 21 in Figure 2.2, illustrate the registration process of two car-specific services `CarOemProxyService` and `CarPropertyService`. Note that some car-services are registered directly using `CarLocalServices.addSerivce` – i.e., without `constructWithTrace` in the Constructor (e,g, `CarUserService`, Line 24).

Car-Specific Permissions in AAOS. To protect access to sensitive car functionality, car services may check for *car-specific* permissions, which are defined centrally by AAOS in a dedicated configuration file *packages/modules/Permission/tests/cts/permissionpolicy/res/raw/automotive_android_manifest.xml*. The majority of these permissions (135 out of 145) are signature-level, meaning they can only be granted to apps signed with the same certificate as AAOS or as the automaker firmware. Other traditional access control features, such as UID and PID checks, can be individually or jointly checked by car services to protect underlying resources. This ensures that only trusted apps or system components can access sensitive car-related services, thereby enhancing security and control over automotive features.

2.4.2 Automotive Entry Points in Traditional Services

Adopting Android to AAOS further involves customizing certain traditional entry points (APIs). Specifically, some APIs check if the platform is automotive (i.e., via checking if `PackageManager.FEATURE_AUTOMOTIVE`) and accordingly provide different functionality handling, which can be largely categorized into two different categories: (1) implement different logic and (2) block functionality.

Example of Logic Customization: Figure 3.1 depicts an implementation of the traditional entry point `setMasterMute(..)` in `AudioService`. As shown in Line 17, the API enforces that the calling user should be `USER_SYSTEM` if the device is automotive, hence ensuring that non-privileged users (such as secondary and guest users) cannot manipulate audio settings in a car environment.

Example of Functionality Blocking: Figure 2.4 depicts a simplified implementation of another traditional entry point `setLocationEnabled(..)` in `DevicePolicyManager-`

```
// Third-party app
public class App{
    Car mCar = Car.create(context);
    CarPropertyManager pm = (CarPropertyManager) mCar.getCarManager(Car.
        PROPERTY_SERVICE);
    pm.getProperty(propertyId, zoneId);
    pm.getPropertyList();
    ....
}
// ICarProperty.aidl
interface ICarProperty {
    CarPropertyConfigList getPropertyList();
    CarPropertyValue getProperty(int prop, int zone);
    ...
}
// CarPropertyService.java
public class CarPropertyService extends ICarProperty.Stub implements
    CarServiceBase, PropertyHalService.PropertyHalListener {
    ...
    public CarPropertyValue getProperty(int propertyId, int areaId)
        throws IllegalArgumentException, ServiceSpecificException {
        validateGetParameters(propertyId, areaId);
        ...
    }
    public CarPropertyConfigList getPropertyList() {
        int[] allPropIds;
        ...
    }
    ...
}
```

Figure 2.3: An example implementation of how to access a specific car service, such as `CarPropertyService` from an app.

`Service`. As shown in Lines 4-6, when an automotive system requests to disable location, the API ignores the request. For non-automotive systems or when enabling location on automotive devices, the method adjusts the location setting for the user. Hence, the API ensures that in automotive builds, calls to disable location services are ignored to ensure critical features like navigation and emergency assistance remain functional. Disabling these services could compromise safety and the overall driving experience.

```
1  @Override
2  public void setLocationEnabled(ComponentName who, boolean locationEnabled
       ) {
3      ...
4      if (mIsAutomotive && !locationEnabled) {
5          Slogf.i(LOG_TAG, "setLocationEnabled(%s, %b): ignoring for user %
               s on automotive build", who.flattenToShortString(),
               locationEnabled, userHandle);
6          return;
7      }
8      mInjector.binderWithCleanCallingIdentity(() -> {
9          boolean wasLocationEnabled = mInjector.getLocationManager().
               isLocationEnabledForUser(userHandle);
10         Slogf.v(LOG_TAG, "calling locationMgr.setLocationEnabledForUser(%
               b, %s) when it was %b",
11                 locationEnabled, userHandle, wasLocationEnabled);
12         mInjector.getLocationManager().setLocationEnabledForUser(
               locationEnabled, userHandle);
13     ...
14     }
15 }
```

Figure 2.4: Simplified Implementation of `DevicePolicyManagerService.setLocationEnabled` retrieved from Android **version 13** - *shows functionality blocking for automotive builds.*

2.4.3 AAOS Vehicle Properties

To support vehicle property access, the AAOS integration includes a class for vehicle properties, listing property IDs for vendors to map with their supported vehicle properties in the native hardware layer. For instance, in the latest version 14 release, there are 259 property IDs defined. These IDs are used with `CarPropertyManager` APIs, such

as `CarPropertyManager. getProperty(int, int)` and `CarPropertyManager. setProperty(Class, int, int, Object)` [14]. Vendors can also add their customized property IDs to this list as long as they abide by the guidelines in [13]. For these vehicle properties, specific permissions for each property are defined. These are defined in 2 permission mappings classes: 1) *PropertyPermissionMapping* [10] for AOSP defined property IDs and 2) *VehicleVendorPermission* [15] for vendor defined property IDs.

Chapter 3

Motivation

The rapid and uncertain expansion of Android into AAOS provides the motivation of our work. In this section, we discuss concrete examples.

3.1 Uncertainty in AOSP Automotive Restriction Implementations

AAOS developers at times lack explicitly-stated functionality and security specification. Through our manual inspection of AAOS source code, we observe that they may not have accurate knowledge of which functions or original Android pieces of code need modifications to properly integrate AAOS. For example, as illustrated in the code snippets in Figure 3.1 (Lines 7-9), AAOS engineers are uncertain about the necessity of the Platform Automotive check when using fixed volume. The check is noted as "*to be removed*" in version 9 but is still lingering in the latest version 14. Although a change/update in the implementation was depicted, that is, in the earlier versions from 9 to 13 (Figure 3.1), the check resulted in returning back to the internal method caller without proceeding any further. However, in version 14, as shown in Figure 3.2 in line 11, the return statement was replaced with the local variable `mute` to be explicitly set to `False`. But yet, no changes were made to the check or its documentation, indicating ongoing confusion. Figure 3.3 shows clear visual differences in the code implementations.

```java
public void setMasterMute(boolean mute, int flags, String callingPackage,
    int userId, String attributionTag) {
    enforceModifyAudioRoutingPermission();
    if (DEBUG_VOL) {
        Log.d(TAG, String.format("Master mute %s, %d, user=%d", mute,
            flags, userId));
    }
    if (!isPlatformAutomotive() && mUseFixedVolume) {
        // If using fixed volume, we don't mute.
        // TODO: remove the isPlatformAutomotive check here.
        // The isPlatformAutomotive check is added for safety but may not
            be necessary.
        return;
    }
    // For automotive,
    // - the car service is always running as system user
    // - foreground users are non-system users
    // Car service is in charge of dispatching the key event include
        global mute to Android.
    // Therefore, the getCurrentUser() is always different to the
        foreground user.
    if ((isPlatformAutomotive() && userId == UserHandle.USER_SYSTEM)
            || (getCurrentUserId() == userId)) {
        if (mute != AudioSystem.getMasterMute()) {
            AudioSystem.setMasterMute(mute);
            sendMasterMuteUpdate(mute, flags);
        }
    }
}
```

Figure 3.1: Simplified Implementation of `AudioService. setMasterMute` retrieved from Android **version 13** *(Depicting Uncertainty in Implementing Automotive Restrictions)*.

```
public void setMasterMute(boolean mute, int flags, String callingPackage,
    int userId, String attributionTag) {
    enforceModifyAudioRoutingPermission();
    if (DEBUG_VOL) {
        Log.d(TAG, TextUtils.formatSimple("Master mute %s, flags 0x%x,
            userId=%d from %s",
                mute, flags, userId, eventSource));
    }
    if (!isPlatformAutomotive() && mUseFixedVolume) {
        // If using fixed volume, we don't mute.
        // TODO: remove the isPlatformAutomotive check here.
        // The isPlatformAutomotive check is added for safety but may not
            be necessary.
        mute = false;
    }
    // For automotive,
    // - the car service is always running as system user
    // - foreground users are non-system users
    // Car service is in charge of dispatching the key event include
        global mute to Android.
    // Therefore, the getCurrentUser() is always different to the
        foreground user.
    if ((isPlatformAutomotive() && userId == UserHandle.USER_SYSTEM)
            || (getCurrentUserId() == userId)) {
        if (mute != mMasterMute.getAndSet(mute)) {
            sVolumeLogger.enqueue(new VolumeEvent(
                    VolumeEvent.VOL_MASTER_MUTE, mute));
            mAudioSystem.setMasterMute(mute);
            sendMasterMuteUpdate(mute, flags);
        }
    }
}
```

Figure 3.2: Simplified Implementation of `AudioService. setMasterMute` retrieved from Android **version 14** *(Depicting Uncertainty in Implementing Automotive Restrictions)*.

(a) v9 - v13 (a) v14

Figure 3.3: Code Difference of `AudioService. setMasterMute` retrieved from Android **version 13 and 14** *(Depicting Uncertainty in Implementing Automotive Restrictions)*.

3.2 Anomaly in Car Service API in AOSP

Access Controls in APIs ensure that only authorized users can access specific resources. However, these implementations can have flaws, such as inconsistencies [17, 65, 21], outdated permissions, misplaced position, and other anomalies, which can lead to security risks. We depicted such a flaw in the access control of an API in the `CarProperty-Service`. As shown in Figure 3.4 and Figure 3.5, an anomaly was identified in the `CarPropertyService`, specifically within the `getPropertyList` API, which allows for partial bypass of an access control mechanism. This API is responsible for returning a list of property configurations that the caller is authorized to access. Under normal circumstances, if the caller does not have the necessary permissions, the API returns an empty list, preventing unauthorized users from discovering the available property IDs in the car configuration. However, a related API, `getProperty`, exposes a log during its execution that can be leveraged to infer the property IDs present in the configuration. This occurs despite the caller lacking the appropriate permissions. The vulnerability arises in `getProperty` since the access control is enforced after checking and logging the property ID, unintentionally exposing sensitive information in the log. The correct approach to mitigate this issue would be to apply access control checks before performing any opera-

tions, such as verifying the existence of the property ID or logging information. In the case of `getProperty`, access control enforcement should be positioned before the property ID check to prevent unauthorized disclosure of configuration data. The case was reported to Google who promised to remediate it in future versions.

Such instances of developer confusion and observed anomaly have steered our focus toward identifying potential access control issues within the largely understudied AAOS.

```
1  // Return the list of properties' configs that the caller may access.
2  public List<CarPropertyConfig> getPropertyList() {
3      int[] allPropId;
4      // Avoid Perm checking under lock.
5      synchronized (mLock) {
6          allPropId = new int[mConfigs.size()];
7          for (int i = 0; i < mConfigs.size(); i++) {
8              allPropId[i] = mConfigs.keyAt(i);
9          }
10     }
11     // Cache the granted Perms
12     Set<String> grantedPerm = new HashSet<>();
13     List<CarPropertyConfig> availProp=new ArrayList<>();
14     if (allPropId == null) {
15         return availProp;
16     }
17     for (int propId : allPropId) {
18         String readPerm=getReadPerm(propId);
19         String writePerm=getWritePerm(propId);
20         if (readPerm == null && writePerm == null) {
21             continue;
22         }
23         // Check if context already granted Perm first
24         if (checkAndUpdateGrantedPermSet(mContext, grantedPerm, readPerm) ||
               checkAndUpdateGrantedPermSet(mContext, grantedPerm, writePerm)) {
25             synchronized (mLock) {
26                 availProp.add(mConfigs.get(propId));
27             }
28         }
29     }
30     ...
31     return availProp;
32 }
```

Figure 3.4: Simplified Implementation of `CarPropertyService. getPropertyList()` retrieved from Android **version 13**.

```
@Override
public CarPropertyValue getProperty(int prop, int zone) throws
    IllegalArgumentException, ServiceSpecificException {
     synchronized (mLock) {
         if (mConfigs.get(prop) == null) {
             // Do not attempt to register an invalid propId
             Slogf.e(TAG, "getProperty: propId is not in config list:0x" +
                 toHexString(prop));
             return null;
         }
     }
     // Checks if android has permission to read property.
     String perm = mHal.getReadPermission(prop);
     if (perm == null) {
         throw new SecurityException("Platform does not have permission to
             read value for "
                 + "property Id: 0x" + Integer.toHexString(prop));
     }
     CarServiceUtils.assertPermission(mContext, perm);
     return mHal.getProperty(prop, zone);
}
```

Figure 3.5: Simplified Implementation of `CarPropertyService. getProperty(..)` retrieved from Android **version 13**.

Chapter 4

Our Approach & Tackling Challenges

In this section, we briefly outline our approach and address the key challenges.

4.1 Our Approach

We introduce our approach, *AutoAcRaptor*, for evaluating AAOS integration. Our approach consists of four phases which aim to achieve the following two high-level goals:(1) generate access control specifications for all APIs by analyzing AAOS codebases , and (2) leverage the specification to derive potential anomalies within automotive APIs. Throughout the process, it combines static analysis techniques and natural language processing (NLP) methods.

Figure 4.1 illustrates a breakdown of the steps in our approach. After identifying car system services and corresponding entry points in a given AAOS ROM (steps 1 and 2), *AutoAcRaptor* proceeds as follows:

1. **Generating Access Control Specifications (Step 3):** In this stage, *AutoAcRaptor* looks for access control checks used by Android to protect sensitive resources. The checks span traditional checks (e.g., permission enforcements, UID, PID, physical user checks) as well as custom checks used exclusively in AAOS (e.g., automotive feature checks enforced to customize or block access to API functionality (as discussed in Chapter 2). *AutoAcRaptor* relies on a set of compiled static patterns to automatically

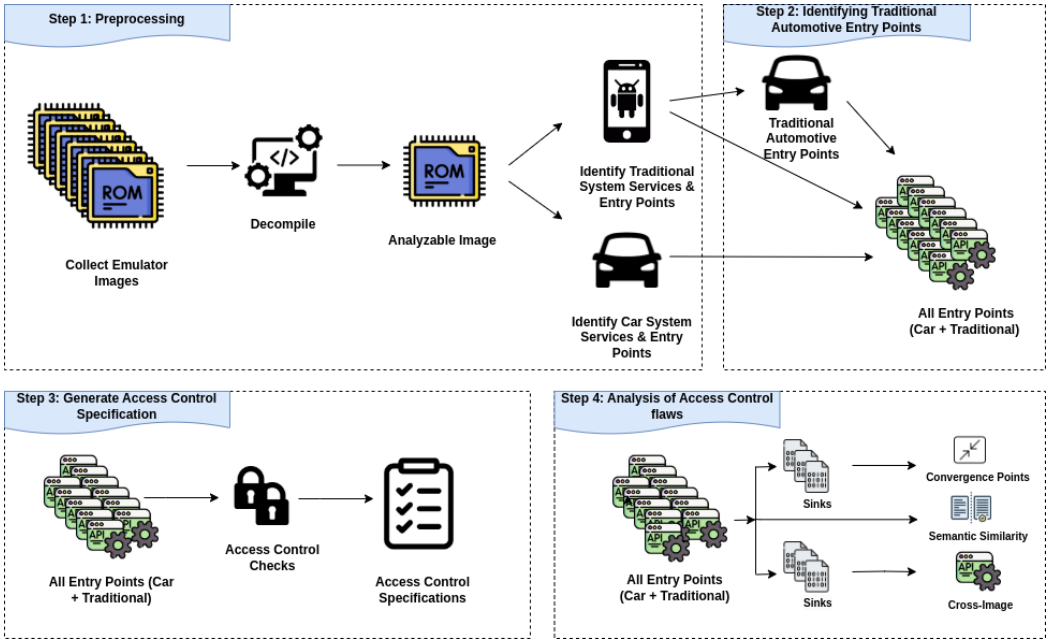

Figure 4.1: Overview of Access Control Anomaly Detection Methodology in AAOS.

identify access control specification in API implementations.

2. **Evaluating AAOS API implementations (Step 4):** After generating the access control specification, *AutoAcRaptor* evaluates its correctness in an effort to detect potential anomalies. Due to the lack of an oracle (e.g., a formal, written security specification in the traditional software engineering sense), we cannot easily determine whether the access control implementation in the automotive-related APIs (as derived in *step 3*) sufficiently protect underlying resources. Hence, we resort to approximate solutions that compare access control enforced along multiple instances of the same resource. If a resource can be reached through two APIs with differing access control implementation, *AutoAcRaptor* considers the case as a potential anomaly. Next, we describe the approximate solutions *AutoAcRaptor* relies on to detect APIs leading to similar resources:

   (a) **Convergence Analysis:** *AutoAcRaptor* conducts traditional convergence analysis across different APIs [65, 17, 37] to identify those overlapping in functionality. This solution assumes that if the same sink (e.g., method invocation

instruction with similar args, global variable update or read) is reachable by two APIs, then they must be functionally similar, hence should enforce similar access control.

```
1  // API 1
2  public void registerClusterStateListener(IClusterStateListener listener)
       {
3      enforcePermission(Car.PERMISSION_CAR_INSTRUMENT_CLUSTER_CONTROL);
4      if (!mServiceEnabled) throw new IllegalStateException("Service is not
           enabled");
5      mClientListeners.register(listener);
6  }
7  // API 2
8  public void registerClusterNavigationStateListener(
       IClusterNavigationStateListener listener) {
9      enforcePermission(Car.PERMISSION_CAR_MONITOR_CLUSTER_NAVIGATION_STATE
           );
10     if (!mServiceEnabled) throw new IllegalStateException("Service is not
           enabled");
11     mClientNavigationListeners.register(listener);
12 }
```

Figure 4.2: Simplified Implementation of `ClusterHomeService.registerClusterStateListener(..)` and `registerClusterNavigationStateListener(..)` retrieved from Android **version 14**.

(b) **Similarity Analysis:** Unlike convergence analysis where an exact common sink is required to identify functionally-overlapping APIs, this solution adopts a more relaxed assumption. It relies on semantic similarity analysis to find such APIs – for example, it considers APIs exhibiting similar names (e.g., *CarPropertyService. getProperty* and *CarPropertyService. getPropertyList*) (as discussed in the motivation section) to provide related functionality, and thus should implement the same/similar access control. This analysis can cover APIs missed by convergence analysis. Another example would be, the `registerClusterStateListener` and `registerClusterNavigationStateListener` APIs (in Figure 4.2) both manage cluster-related listeners with similar permission checks. Note, such APIs do not perform the same function but very similar ones, hence they either require the same permission or similar permissions. These APIs might be overlooked in the convergence analysis, thus we use this lightweight analysis to identify them

for further inspection to have a comprehensive study of the security evaluation of similar named APIs as well.

(c) **Cross-Image Analysis of Custom APIs:** For the custom images, *AutoAcRaptor* is able to identify APIs that originate from AOSP itself (same API method signature) but have been modified in its implementation. *AutoAcRaptor* compares these modified AOSP API implementations with its corresponding API version in the original AOSP as well as with its corresponding API version in other custom vendor images of the same android version. This helps identify any changes or differences both in the API functionality and the access control implementations.

4.2 Tackling Challenges

In this section, we present a brief overview of the challenges encountered in our study and our strategies for addressing them.

Collecting AAOS Images *(Challenge 1).* Although AAOS source code (Google version) is readily available on AOSP repositories [5], it is more difficult to obtain automaker-customized AAOS code. Specifically, (1) we cannot extract ROMs (for decompilation) from physical vehicles due to obvious cost and availability reasons, and (2) to the best of our knowledge, there are no public repositories for automaker AAOS source code. To address this challenge, we resort to collecting available AAOS *emulator* images provided by some vendors through the official Android Documentation webpage [30] and the vendors' own websites [6, 7, 9, 16]. Since these images are provided for developers to test their car applications, these collected modern emulators closely mimic the real vehicular devices, providing a reliable environment for our analysis. More details on the collected images are discussed in Chapter 5.

Identifying Car-Specific AAOS Entry Points *(Challenge 2).* We observe that AAOS entry points are defined in the framework using non-conventional patterns, as their corresponding system services are not registered in the *ServiceManager* in the usual way. As such, we cannot rely on the static patterns traditionally used by prior work [17, 65, 22, 37, 21], in order to recover the car-specific system services and their entry points.

AutoAcRaptor addresses this challenge by encoding the new patterns we observed for registering car entry points (refer to Figure 2.3 and Chapter 2). More details about the patterns are discussed in Chapter 5.

Identifying Car-Specific Customization in the Traditional APIs *(Challenge 3).* We observe that not all original Android APIs are customized for AAOS integration (Refer to Table 6.1 for a breakdown of entry points in the studied ROMs). To scale up our security analyses, it is essential to narrow down our investigation to those modified APIs, i.e., entry points that are modified by Google or automakers to support automotive functionality. *AutoAcRaptor* adopts a few rules to filter such APIs. More details are discussed in Chapter 5.

Identifying and Extracting Primary Sinks for Convergence Analysis *(Challenge 4).* To accurately identify functionally-overlapping APIs, our analysis should be able to select those converging on"*primary*" sinks. This task is inherently challenging as shown by prior work [22, 65, 21, 17, 32]. We resort to a few strategies to tackle the challenge that we discuss elaborately in section 5.3.

Chapter 5

System Design

In this section, we dive into the design details of *AutoAcRaptor*.

5.1 Phase I & II: Preprocessing & Identifying Entry Points

5.1.1 Collecting & Decompiling AAOS ROMs

In the first phase *(Step 1)*, *AutoAcRaptor* collects, decompiles, and pre-processes target AAOS ROMs for security analysis. The ROMs may span Google and/or custom ROMs. Preprocessing extracts framework and car-specific libraries, using a variety of tools (apktool [2], dex2jar [4], smali [12], etc.

5.1.2 Identifying and Extracting Car Entry Points

AutoAcRaptor first identifies the `ICarImpl` class (the class that implements the `CarService`, as discussed in Chapter 2) and its constructor. It then identifies the method invocations - 1) `constructWithTrace`, and 2) `CarLocalServices.addService` (recall that some services are directly registered using `CarLocalServices.addService`, necessitating *AutoAcRaptor* to consider it individually along with `constructWithTrace` to ensure comprehensive

coverage). *AutoAcRaptor* resolves the arguments of these invocation calls through precise data-flow analysis to retrieve the concrete type of the car service classes. Finally, it finds the exposed interface classes of each of the car services similar to the traditional android system services (discussed in the following subsection), and retrieves their declared public APIs.

5.1.3 Identifying and Extracting Traditional Entry Points

To identify standard system services (i.e., those shared with non-AAOS Android, such as *PackageManagerService* and *ActivityManagerService*), we rely on traditional registration patterns as performed by prior work [22, 65, 21, 17]. Specifically, *AutoAcRaptor* first pinpoints invocations to `publishBinderService` and `addService` methods, which are used to register an Android system service in the `ServiceManager` registry. It resolves the service name constant passed to these methods and identifies the concrete service classes. Then it locates the exposed interface class for each service and extracts all declared public APIs implemented by the system services.

5.1.4 Filtering Automotive-related entry points from Traditional Entry Points

Here we aim to filter entry points from traditional services that have been customized by Google (or automakers) to accommodate AAOS. Through our manual analysis of AAOS source code, we observed that such customization exhibits two distinct footprints:

Automotive keywords: *AutoAcRaptor* marks APIs with *"automotive"* keywords in them, including in the API name, used variable names, invoked inner method names, and in recovered constant values. Additionally, it identifies other relevant keywords and synonyms retrieved from AAOS documentation, such as "auto", "car", and "vehicle". This simplistic approach allows us to easily identify the most of the relevant automotive-related APIs from the traditional system service APIs. Figure 5.1 illustrates a simplified implementation of an API `DevicePolicyManagerService.setPasswordQuality` tagged as automotive-related by *AutoAcRaptor*, as it is calling an Internal method (`notSupportedOnAutomotive` - line 4) with an automotive keyword in it.

```
1  // API in DevicePolicyManagerService
2  public void setPasswordQuality(ComponentName who, int quality, boolean
       parent) {
3      if (!mHasFeature || notSupportedOnAutomotive("setPasswordQuality")) {

4          return;
5      }
6      ....
7  }
8  // private internal method
9  private boolean notSupportedOnAutomotive(String method) {
10     if (mIsAutomotive) {
11         Slogf.i(LOG_TAG, "%s is not supported on automotive builds",
               method);
12         return true;
13     }
14     return false;
15 }
```

Figure 5.1: Simplified Implementation of `DevicePolicyManagerService.setPasswordQuality` retrieved from Android **version 14** - *Depicts calling an Internal method (*`notSupportedOnAutomotive` - *Line 3) with automotive keyword* .

Feature checks: *AutoAcRaptor* similarly marks APIs performing an automotive feature check. Specifically, we inspect conditional statements in the API method body and flag those where one of the predicates leads to the invocation of `hasSystemFeature`, a method in the `PackageManager` class for retrieving information about installed application packages. *AutoAcRaptor* resolves the feature name to detect the string: `"android.hardware.type.automotive"`, which is used to check if the system is automotive Figure 5.2 illustrates a simplified implementation of an API `LockSettingsService.addWeakEscrowToken` tagged as automotive-related by *AutoAcRaptor*, since it is calling an Internal method (`checkManageWeakEscrowTokenMethodUsage` - Line 7) that has an automotive system feature check (Line 11).

```
1  // API in LockSettingsService
2  public long addWeakEscrowToken(byte[] token, int userId,
       IWeakEscrowTokenActivatedListener listener) {
3      checkManageWeakEscrowTokenMethodUsage();
4      ...
5  }
6  // private internal method
7  private void checkManageWeakEscrowTokenMethodUsage() {
8      mContext.enforceCallingOrSelfPermission(
9              Manifest.permission.MANAGE_WEAK_ESCROW_TOKEN,
10             "Requires MANAGE_WEAK_ESCROW_TOKEN permission.");
11     if (!mContext.getPackageManager().hasSystemFeature(PackageManager.
           FEATURE_AUTOMOTIVE)) {
12         throw new IllegalArgumentException(
13                 "Weak escrow token are only for automotive devices.");
14     }
15 }
```

Figure 5.2: Simplified Implementation of `LockSettingsService.addWeakEscrowToken` retrieved from Android **version 14** - *Depicts calling an Internal method (Line 7) that has automotive feature check in Line 11* .

5.2 Phase III: Generating Access Control Specification

In this phase, *AutoAcRaptor* generates a comprehensive Access Control Specification, inspired by prior work [65, 19, 17, 22], in a path-insensitive manner. *AutoAcRaptor* automatically determines the security checks performed by an entry point, such as the permissions it enforces, and the IDs, packages, and features it checks.

Permission Check Methods. In Android's framework source code, permissions are represented as string constants. When conducting permission checks, these strings are passed to a check method. These check methods follow a common naming pattern such as they start with "enforce", "check", or "validate" and end with "Permission" (e.g., checkCallingOrSelfPermission, enforceCallingPermission, validatePermission) (as discussed by [65]). *AutoAcRaptor* automatically identifies such permission methods and resolves its arguments to identify specific permissions being checked.

Security Checks Using Identifiers. Apart from permission checks, *AutoAcRaptor* can

also identify other types of security checks automatically. These checks depend on specific conditions, often based on identifiers such as application IDs, process IDs, UIDs, user IDs and package names. These checks are identified within conditional statements in the body of a method. Similar to the permission check methods (as discussed in the previous section), some of these identifier methods start with "getCalling" and mostly end with "ID" or "id" (e.g., getCallingUid, getCallingPid, getCallingUserId).

Automotive Feature Check Methods. Additionally, *AutoAcRaptor* detects another category of checks. As discussed in Chapter 2, we observed in the AAOS source code that some APIs block functionality for automotive devices. Thus, when *AutoAcRaptor* identifies the condition for automotive build checks, it specifically checks for `hasSystemFeature` invocations and resolves its arguments to determine if the feature check is for automotive. It also identifies two wrappers of `hasSystemFeature`: 1) a method called `isPlatformAutomotive` and 2) a boolean variable `mIsAutomotive` which are often used in AAOS.

5.3 Phase IV: Analysis using the Access Control Specification

Next, in this phase, we aim to evaluate automotive entry points for anomalies using the Access Control Specification through three methods: Convergence Analysis, Similarity Analysis, and Cross-Image Analysis.

5.3.1 Convergence Analysis

As discussed in Chapter 4, *AutoAcRaptor* needs to 1) identify and extract potential sinks of the entry points, 2) identify pairs of entry points that converge to the same primary sink and then 3) evaluate their access controls to detect potential anomalies.

Identifying and Extracting Potential Sinks. For potential sink extraction, we use a combined technique of static analysis with natural language processing (NLP). We (1) rely on frequency analysis to filter out commonly invoked instructions in APIs (e.g., logging statements as `Log.d(..)`, string construction statements like `StringBuilder.append(..)`), (2) remove corresponding data-dependent instructions, i.e., those exclusively related to

frequent instructions, and (3) focus on instructions that are highly related to the API functionality through NLP techniques. We briefly elaborate on these steps here:

1. **Frequency Analysis:** *AutoAcRaptor* first performs a frequency analysis, i.e., it generates an *instruction-to-API frequency* mapping. It identifies all the instructions from all the identified APIs in a ROM and then tracks how many APIs use the instruction in their implementation. This instruction-to-API frequency mapping allows *AutoAcRaptor* to filter out instructions that are invoked excessively across multiple APIs. By applying frequency analysis, *AutoAcRaptor* reduces noise by discarding these less relevant instructions, achieving a significant reduction. Our analysis of the frequency mapping reveals that approximately 12% of the instructions are the most commonly used by the APIs and are the least likely to be potential sinks. Additionally, *AutoAcRaptor* uses a manually populated blacklist to filter out instructions that have minimal relevance to the core functionality of the API. Although these instructions should be filtered out during frequency analysis, the blacklist serves as a comprehensive measure. This list includes access control and feature check instructions, as their primary role is to control access to sinks or verify feature support, making them unlikely to represent primary sinks.

2. **Removal of Data-Dependent Instructions:** To identify and remove instructions that depend on those filtered in the previous step, *AutoAcRaptor* performs precise data flow analysis (using *def-use chains*) further narrowing down the potential sinks.

3. **API-Relevant Instructions Using NLP:** *AutoAcRaptor* employs two criteria to pinpoint potential sinks. First, it examines the parameters passed to the API and checks if any of the filtered instructions use these parameters as arguments. Instructions that utilize at least one API parameter are considered as potential sinks. Second, it applies a similarity-based filter to assess the similarity of each instruction (i.e., method and field name) with the API name. *AutoAcRaptor* leverages *UniXcoder* [39] to compute this similarity between two strings. *UniXcoder* is a unified cross-modal pre-trained NL-PL model, for generating word embeddings. The model's tokenizer processes the API name M and associated instruction I, creating their embeddings. We then use the cosine similarity function f to measure the similarity between *embedding (M)* and *embedding (I)*. By applying an empirically determined threshold, we assess the degree of match between the API and the instruction, filtering out less relevant matches.

For our similarity analysis in sink extraction and API-name comparison, we experimented with four models, including UniXcoder, which were: Jaccard (a similarity coefficient that measures the similarity between two sets by dividing the size of their intersection by the size of their union), transformer (a neural network architecture for handling sequential data), and an enhanced version of tranformer - Sentence_transformer (a framework for embedding sentences and text into fixed-size vectors, suitable for tasks like semantic search, clustering, and classification). We tested with multiple and randomly selected APIs and our findings revealed that UniXcoder performed the best in our context. Further details on the threshold value selection for UniXcoder are provided in Chapter 6 in RQ4.

Finally, using these filtering techniques, *AutoAcRaptor* achieves a reduction of approximately 92% of instructions on an average per entry point, narrowing down to the most relevant instructions/ primary sinks of an API.

Identifying Converging Entry Point pairs: After *AutoAcRaptor* generates the access control specification and identifies potential sinks for each API, it stores the access control instructions and the sink instructions in a textual format - including the relevant meta-data of each instruction. *AutoAcRaptor* performs a pairwise convergence analysis to evaluate if two APIs share common sinks. This involves comparing the potential sinks of a pair of APIs within the image to determine if they converge on identical sink points, either by having the same sinks or by sharing at least one common sink instruction. To do this, *AutoAcRaptor* compares the textual sink instruction of two APIs to identify instructions that overlap. Any matching of instructions will flag the APIs pairs as converging.

Access Control Evaluation of Converging Entry Points: When *AutoAcRaptor* identifies a converging pair, it examines the access control mechanisms of the entry points, referring to the previously generated access control specifications. If *AutoAcRaptor* detects that the pairs converge on a common sink but have different access control implementations, it flags the pair as potentially anomalous.

5.3.2 Similar API Analysis

AutoAcRaptor uses *UniXcoder* to identify in-image entry point pairs with semantically similar method signatures, i.e., considering both the entry point method name and its

service class name. It flags pairs with a high semantic similarity score (0.8 or higher which was determined empirically - *discussed in Chapter 6 RQ4*). Then, as discussed in the convergence analysis step, it evaluates their access controls and flags any pairs with differences in them as potentially anomalous.

5.3.3 Cross-Image Analysis

For the cross-image analysis, *AutoAcRaptor* first identifies the entry points that are custom or vendor introduced. It identifies 1) New entry points and 2) modified AOSP entry points (as discussed in Chapter 4). And the latter ones are of the main focus for this analysis. *AutoAcRaptor* identifies the modified AOSP entry points from all the images in a specific version, for instance, for all "version 12" ROMS, both custom (*GM-Cad-Lyriq-SUV-24 (V12L)* and *Honda (V12L)*) and AOSP (V12L). It identifies the entry points that are the same in terms of the entry point method signature but differ in the implementation. After spotting these entry points across images in the same version, *AutoAcRaptor* evaluates their access controls using the generated access control specification (discussed in Section 5.3) and flags the entry points if they have different access control implementations.

Chapter 6

Evaluation

We developed *AutoAcRaptor*, a prototype built on top of the WALA static analysis library [60] and enhanced with UniXCoder [39] for NLP-based similarity and sink analysis. The tool analyzes Android automotive framework bytecode to identify entry points, generate access control specifications, and perform three analyses: Convergence, Similarity, and Cross-Image, as detailed in Chapter 4 and Chapter 5. It also highlights newly introduced and modified AOSP APIs and identifies differences in access control checks within custom images. Our evaluation was conducted on a machine with an 8-core Intel Core-i7 processor, featuring 12 cores with a maximum clock speed of 5.0 GHz per core, and 16GB RAM.

6.1 Target of Study

We collect two AAOS AOSP and ten custom AAOS ROMs of Android versions 9 through 14 from vendors like General Motors (GM) [6], Honda [7], Volvo [16] and Polestar [9]. As outlined in Chapter 5, we use emulators to collect vendor ROMs and decompile them using various tools. Particularly, we had to discard two ROMs - Volvo v10 and Polestar v9 - because we found their decompiled files to be excessively corrupted. We evaluate *AutoAcRaptor* on the rest, which amounts to two AAOS AOSP and eight custom vendor AAOS ROMs. The details of these ROMs are provided in the first column of Table 6.1. Note that in our analyzable set of images, we have two v11 images (from GM and Honda) that exhibited an unusually low number of traditional APIs. The issue appears to be due to the decompilation process, where the interfaces of the system service classes were not properly decompiled. We still keep them and consider them for further analysis as our tool correctly detected the extracted car-specific system services and APIs which is still

relevant for our study.

6.2 Research Questions

To evaluate our solution, we answer the following research questions:

- **RQ1:** What is the extent of Android customization for AAOS?
- **RQ2:** What is the access control landscape in AAOS entry points?
- **RQ3:** Does AAOS customization introduce potential access control anomalies?
- **RQ4:** What is the accuracy of *AutoAcRaptor*?
- **RQ5:** What is the performance / practicality gain of *AutoAcRaptor*?

6.2.1 RQ1: What is the extent of Android customization for AAOS?

To address this research question, we identify the number of car-specific system services and entry points, along with analyzing automaker customization patterns.

Table 6.1: Statistics of the Android Automotive Images.

	System Services		Entry Points					Customized Entry Points	
OS Image	**Traditional**	**Car**	**TOTAL Entry Points**	**Traditional Entry Points**	**Car Entry Points**	**Broadcast Receivers**	**Traditional Automotive Entry Points**	**NEW Custom Entry Points**	**MODIFIED AOSP Entry Points**
GM-SUV-22 (V10)	165	31	3163	2958	145	60	55	249	217
Polestar 2 (V10)	142	30	2908	2724	135	49	55	0	217
GM-Cad-Lyriq-SUV-23 (V11)	159	30	3295	3070	167	58	53	5	231
GM-SUV-23 (V11)	133	30	228	38	156	34	0	1	40
Honda (V11)	124	29	180	24	135	21	0	2	31
Volvo-XC40 (V11)	151	29	3254	3057	156	41	53	4	223
GM-Cad-Lyriq-SUV-24 (V12L)	165	38	2233	1908	259	66	50	20	38
Honda (V12L)	163	36	2214	1909	244	61	50	1	38
AOSP (V13)	177	40	3634	3254	261	119	64	-	-
AOSP (V14)	168	44	2285	1871	339	75	65	-	-

Number of car specific system services. Columns 2 and 3 in Table 6.1 list *traditional* and *car* system services, respectively. On average, *AutoAcRaptor* identifies 155 traditional

system services and 34 car system services. The number of car system services reaches up to 38 in the custom image - GM-Cad-Lyriq-SUV- 24 (V12L) and up to 44 in the AAOS AOSP image version 14.

Number of custom (new and modified) entry points in AAOS (Android OS to AAOS). Since AAOS is a tailored version of Android OS, here we report the number of entry points in each image. We categorize the entries into - *Traditional Entry Points*, *Car Entry Points*, *Broadcast Receivers* , *Traditional Automotive Entry Points* and *Customized Entry Points (by automakers - which are further categorized into two categories - NEW Custom Entry Points and MODIFIED AOSP Entry Points*). Columns 4-8 *(Entry Points)* in Table 6.1 show all the entry points found in the analyzed ROMs by *AutoAcRaptor*. On an average, each ROM exhibited about more than 2300 entry points in total (*TOTAL Entry Points*), except for the partially corrupted ROMs - GM-SUV-23 (V11) and Honda (V11). Though the number of traditional APIs (*Traditional Entry Points*) vary significantly, the number of traditional automotive APIs (*Traditional Automotive Entry Points*) remain consistent across versions and vendors. The car APIs (*Car Entry Points*), on the other hand, show a consistent increase in each new version. On average, 1.4 % of traditional automotive entries are introduced in *LocationManagerService*, 81.1 % are introduced in *AccountManagerService*, 13.8 % are introduced in *LockSettingsService*, 1.1 % are introduced in *AudioService*, 2.1 % are introduced in *DevicePolicyManagerService*, and 0.5 % are introduced in *WindowManagerService*. Our list of entry points also includes *Broadcast Receivers*, which exhibit about 58 instances on average per ROM.

Number of custom (new and modified) entry points in AAOS (automaker customized). The last two columns of Table 6.1 (*Customized Entry Points)* show the landscape of customization performed by the automotive vendors on the AAOS entry points. As discussed in Chapter 4 and Chapter 5, we categorize these customizations into 1)*NEW Custom Entry Points* and 2)*MODIFIED AOSP Entry Points*. While the former type of customizations is limited, with GM-SUV-22 (V10) leading in introducing new APIs (about 53% of the customized APIs), the latter is widespread, with GM-SUV-22 (V10), GM-Cad-Lyriq-SUV-23 (V11), Polestar 2 (V10), and Volvo-XC40 (V11) altering significant AOSP APIs - up to 98% of the customizations involve modifications of existing AOSP entry points for these images.

RQ1 Findings Summary. We observed AAOS customization spans about 75% traditional entry points, 2% traditional automotive entry points, 21% car-specific entry points,

5% broadcast receivers, 1.3% NEW entry points introduced by automakers, and 8.3% existing AOSP entry points modified in their implementations by the automakers.

6.2.2 RQ2: What is access control landscape in AAOS entry points?

Table 6.2: Statistics of the Access Controls in the Android Automotive Images

OS Image	**# Entry Points with Access Controls (ALL)**	**# Entry Points with Access Controls (Car)**	**# Entry Points with Access Controls (Traditional Automotive Entry Points)**	**# Entry Points with Access Controls (NEW Custom Entry Points)**	**# Entry Points with Access Controls (Modified AOSP Entry Points by vendors)**
GM-SUV-22 (V10)	1287	73	54	33	105
Polestar 2 (V10)	1254	69	54	0	105
GM-Cad-Lyriq-SUV-23 (V11)	1418	94	52	1	89
GM-SUV-23 (V11)	94	89	0	0	17
Honda (V11)	91	85	0	0	18
Volvo-XC40 (V11)	1422	93	52	1	98
GM-Cad-Lyriq-SUV-24 (V12L)	791	145	50	4	28
Honda (V12L)	789	141	50	0	29
AOSP (V13)	1419	155	61	–	–
AOSP (V14)	1056	211	55	–	–
TOTAL	9621	1155	428	39	489

We report the landscape of access control in AAOS, detailing the number of entry points that enforce an access control in Table 6.2. On average, for each ROM, 41.9% of the entry points featured some form of access control (Column 2 - *# Entry Points with Access Controls (ALL)*). Among these, 28% were car-specific entry points (Column 3 - *# Entry Points with Access Controls (Car)*), while 3.8% were traditional automotive entry points (Column 4 - *# Entry Points with Access Controls (Traditional Automotive Entry Points)*). The access control landscape for customized entry points introduced by automakers is illustrated in Columns 5 (*# Entry Points with Access Controls (NEW Custom Entry Points)*) and 6 (*# Entry Points with Access Controls (Modified AOSP Entry Points by vendors)*). These columns indicate that 0.32% of entry points with access controls were newly introduced, while 7.5% were modified AOSP entry points—existing AOSP entry points that vendors have altered. This distribution underscores the varying degrees of customization and the implementation of access controls across different types of entry points within the system.

6.2.3 RQ3: Does AAOS customization introduce potential access control anomalies?

To address this research question, we investigate two sub-questions: *1) How many entry points lead to likely similar functionality?* and *2) How many of those are potentially anomalous / flawed?* We perform this for the three analysis strategies *AutoAcRaptor* implements: *1) Convergence Analysis, 2) Similarity Analysis, and 3) Cross-Image Analysis.*

Convergence Analysis. In this analysis, we count how many of the pairs of APIs were detected by *AutoAcRaptor* to be converging to at least one common sink. *AutoAcRaptor* ensures that in these pairs at least one API is automotive (car-specific entry point or traditional automotive entry point). It reports this count in the second column of Table 6.3. In the last column of Table 6.3, it reports the pairs that *AutoAcRaptor* identifies as potentially anomalous due to differences in the access controls. On an average per image, *AutoAcRaptor* reports about 144 pairs to be converging or having overlapping sinks, among which 57% of the pairs are flagged to be anomalous w.r.t to their access control implementations.

Table 6.3: Convergence Analysis report by *AutoAcRaptor*

	Convergence Analysis	
OS Image	**Converging API Pairs (At least ONE Automotive API)**	**Converging API Pairs with Access Control Differences (At least ONE Automotive API)**
GM-SUV-22 (V10)	163	87
Polestar 2 (V10)	160	87
GM-Cad-Lyriq-SUV-23 (V11)	135	105
GM-SUV-23 (V11)	39	17
Honda (V11)	38	17
Volvo-XC40 (V11)	134	105
GM-Cad-Lyriq-SUV-24 (V12L)	183	109
Honda (V12L)	177	104
AOSP (V13)	197	104
AOSP (V14)	221	103

Similarity Analysis. In this analysis, similar to the convergence analysis, we count the number of API pairs detected by *AutoAcRaptor* to be semantically similar in their API names. Again, *AutoAcRaptor* ensures that at least one API in each pair is automotive (car-specific entry point or traditional automotive entry point). The second column of Table 6.4 reports this count. The last column of Table 6.4 shows the pairs that *AutoAcRaptor* identifies as potentially anomalous due to differences in access controls. On an average,

AutoAcRaptor reports that about 55 pairs of APIs have a high similarity in their names, of which around 12% of the pairs are flagged as anomalous.

Table 6.4: Similarity Analysis report by *AutoAcRaptor*

	Similarity Analysis	
OS Image	**Similar API Pairs (At least ONE Automotive API)**	**Similar API Pairs with Access Control Differences (At least ONE Automotive API)**
GM-SUV-22 (V10)	46	1
Polestar 2 (V10)	45	1
GM-Cad-Lyriq-SUV-23 (V11)	45	5
GM-SUV-23 (V11)	39	4
Honda (V11)	30	4
Volvo-XC40 (V11)	44	5
GM-Cad-Lyriq-SUV-24 (V12L)	62	8
Honda (V12L)	62	8
AOSP (V13)	70	10
AOSP (V14)	112	30

Table 6.5: Cross-Image Analysis report by *AutoAcRaptor*

	MODIFIED AOSP Entry Points (Automaker Customized)		
OS Image	**Different Overall Implementation**	**Different Access Control Implementation**	**Same Access Control Implementation**
GM-SUV-22 (V10)	217	3	214
Polestar 2 (V10)	217	3	214
GM-Cad-Lyriq-SUV-23 (V11)	231	3	228
GM-SUV-23 (V11)	40	1	39
Honda (V11)	31	1	30
Volvo-XC40 (V11)	223	3	220
GM-Cad-Lyriq-SUV-24 (V12L)	38	0	38
Honda (V12L)	38	0	38

Cross-image Analysis. In this final analysis, *AutoAcRaptor* delves deeper into the custom images. It identifies existing AOSP entry points that have been modified by automakers, as reported in the second column of Table 6.5. The last two columns show the number of these APIs with differing access control implementations and those with consistent access controls. On an average, *AutoAcRaptor* identifies about 1% of such APIs per image as having differences in access controls, indicating possible anomalies.

6.2.4 RQ4: What is the accuracy of *AutoAcRaptor*?

To determine the accuracy of *AutoAcRaptor* we evaluate it based on its true positive (TP) and false positive (FP) rates. We perform this assessment on a representative ROM - GM-Cad-Lyriq-SUV-24 (V12L), and manually validate all their reported cases in the convergence and similarity analyzes.

Let us first define the true positive and false positive definitions for our convergence and similarity analyses:

Convergence Analysis. If a pair of APIs, converge to a primary sink, that is, an instruction or a set of instructions that depict the main functionality of the APIs, then we consider that pair as a TP. If a pair of APIs, converge to a non-primary sink, then we consider that pair as a FP.

Similarity Analysis. For similarity analysis, we validate whether the two APIs in the pair reported as similar are indeed similar in their functionality. If the APIs depict similar functionality as their name similarity suggests, we consider the reported pair as TP. Otherwise, if the similar named APIs exhibit different functionality, we then consider the pair as FP.

Table 6.6: Impact of Various Similarity Thresholds

	Convergence Analysis		**Similarity Analysis**	
Threshold	**TP**	**FP**	**TP**	**FP**
0.6	0.67	0.33	0.39	0.61
0.7	0.71	0.29	0.41	0.59
0.8	0.71	0.29	0.75	0.25
0.9	0.71	0.29	0.71	0.29

As discussed in Chapter 5, *AutoAcRaptor*'s performance is highly dependent on the similarity score by UniXCoder in the potential sink extraction for convergence analysis and in the API name similarity in the similarity analysis. Thus, *AutoAcRaptor*'s TPs and FPs are dependent on the threshold chosen for the similarity score. Table 6.6 shows *AutoAcRaptor*'s TP and FP rates under multiple threshold values, varying from 0.6 to 0.9. As shown in the table, the threshold 0.8 achieves the best trade-off, leading to lower FPs and higher

TPs. *AutoAcRaptor* achieves a TP rate of about 71% and and a FP rate of about 29% in the convergence analysis and achieves a TP rate of about 75% and and a FP rate of about 25% in the similarity analysis. We manually analyzed about 10% of the randomly selected false positives and found the primary root causes: 1) for the convergence analysis, *AutoAcRaptor* correctly identified a common convergence point but it is not the primary functionality of the APIs, and 2) for the similarity analysis, the APIs are identified as similar based on their names but have no similarity in their behavior, that is, their functionality is significantly different.

Finally, to evaluate the false negative, we analyze of about 5% of entry points pair that were not reported as converging or similar by *AutoAcRaptor*. We did not find any pair that warranted further inspection (i.e., contained at least one convergence point, or was similar), therefore we concluded that *AutoAcRaptor*'s false negative rate is negligible.

6.2.5 RQ5: What is the performance / practicality gain of *AutoAcRaptor*?

Table 6.7: Effectiveness of *AutoAcRaptor*

OS Image	**Manual Effort without *AutoAcRaptor* (API pairs)**	**Manual Effort with *AutoAcRaptor* (API pairs)**	**TIME *AutoAcRaptor* takes for analysis**
GM-SUV-22 (V10)	5000703	93	16m 28s
Polestar 2 (V10)	4226778	93	14m 15s
GM-Cad-Lyriq-SUV-23 (V11)	5426865	114	19m 27s
GM-SUV-23 (V11)	25878	30	2m 7s
Honda (V11)	16110	30	56s
Volvo-XC40 (V11)	5292631	114	17m 26s
GM-Cad-Lyriq-SUV-24 (V12L)	2492028	139	6m 8s
Honda (V12L)	2449791	134	6m 2s
AOSP (V13)	6601161	123	19m 4s
AOSP (V14)	2609470	120	11m 48s

AutoAcRaptor is an automated tool that identifies potentially anomalous API pairs. While it successfully detects these anomalies, determining whether the APIs are genuinely flawed (i.e., vulnerable and exploitable) requires manual inspection.

We run *AutoAcRaptor* on the ROMs from our dataset and measure 1) the reduction in the number of API pairs needed to be analyzed manually, and 2) time taken by *AutoAcRaptor* to analyze each ROM. The complexity of AAOS, along with the lack of accurate documentation on existing exposed entry points necessitates, the developers to compare all possible pairs of entry points to ensure comprehensive coverage. *AutoAcRaptor* automates this comparison through convergence and similarity analysis, resulting in significant reduction of manual efforts. Columns 2 and 3 in Table 6.7 show the number of API pairs that must be analyzed manually by the developers without and with *AutoAcRaptor*. On an average per ROM, *AutoAcRaptor* reduces this number by 99%, narrowing down the pairs for verification significantly to a manageable count.

The last column in Table 6.7 shows the time *AutoAcRaptor* takes to analyze each ROM. As discussed in Section 6.1, we observed that the GM and Honda v11 ROMs did not contain as many traditional APIs as the other ROMs, resulting in *AutoAcRaptor* taking significantly less time in analyzing these ROMs. For the rest, *AutoAcRaptor* takes 17 minutes 26 seconds on an average to complete its analysis.

6.3 Case Studies of Observed Anomalies

To demonstrate the impact of *AutoAcRaptor*, we provide details on a few randomly chosen true positives reported by *AutoAcRaptor* in Table 6.8. We have responsibly reported all the cases to the respective vendors.

Table 6.8: Summary of observed entry points with Anomalies

OS Image	Entry Point	Observed Anomaly
AOSP (V13)	CarPropertyService. getProperty	Reveals protected car configuration information
AOSP (V13)	CarPropertyService. registerListener	Reveals protected car configuration information
AOSP (V13)	DevicePolicyManagerService. setPasswordMinimumLowerCase	Missing Admin Feature check
GM-SUV-22 (V10)	PhoneProjectionBinderService. isProjectionForeground	Reveals foreground projection status
GM-Cad-Lyriq-SUV-24 (V12L)	CarPowerManagementService. unregisterListener	An access control check that restricts an operation to be allowed by system or self only is missing
AOSP (V11)	VibratorService. onExternalVibrationStart	A missing access control check
Honda (V11))	AccountManagerService. getPreviousName	A missing access control check
Volvo-XC40 (V11)	ConnectivityService. startNattKeepaliveWithFd	Not protected. Missing Signature level protection

6.3.1 Case 1: `CarPropertyService`

```
public void registerListener(int propId, float rate, ICarPropertyEventListener listener)
        throws IllegalArgumentException {
    ...
    CarPropertyConfig propertyConfig;
    Client finalClient;
    synchronized (mLock) {
        propertyConfig = mConfigs.get(propId);
        if (propertyConfig == null) {
            // Do not attempt to register an invalid propId
            Slogf.e(TAG, "registerListener:  propId is not in config list: 0x" +
                toHexString(propId));
            return;
        }
        CarServiceUtils.assertPermission(mContext, mHal.getReadPermission(propId));
        ...
}
```

Figure 6.1: Simplified Implementation of `CarPropertyService. registerListener` retrieved from Android **version 13**.

```
1  public void unregisterListener(int propId, ICarPropertyEventListener listener) {
2      ...
3      CarServiceUtils.assertPermission(mContext, mHal.getReadPermission(propId));
4      if (listener == null) {
5          Slogf.e(TAG, "unregisterListener: Listener is null.");
6          throw new IllegalArgumentException("Listener is null");
7      }
8      IBinder listenerBinder = listener.asBinder();
9      float updateMaxRate = 0f;
10     Client client = mClientMap.get(listenerBinder);
11     List<Client> propertyClients = mPropIdClientMap.get(propId);
12     if (mConfigs.get(propId) == null) {
13         // Do not attempt to unregister an invalid propId
14         Slogf.e(TAG, "unregisterListener: propId is not in config list:0x%s", toHexString(
               propId));
15         return;
16     }
17     if ((client == null) || (propertyClients == null)) {
18         Slogf.e(TAG, "unregisterListenerBinderLocked: Listener was not previously "
19                 + "registered.");
20         return;
21     }
22     ...
23 }
```

Figure 6.2: Simplified Implementation of `CarPropertyService. unregisterListener` retrieved from Android **version 13**.

The anomaly observed in the `CarPropertyService getProperty` API is explained in the motivation section already which was promised to be remedied in the future versions. This partial bypass instance because of the permission check being placed after logging instead of before, was also found in two other entry point pairs within the `CarPropertyService` in the `registerListener` and `unregisterListener` APIs. `registerListener` logs the car config information first (lLine 10 in Figure 6.1) and then asserts a permission check (Line 13 in Figure 6.1) but `unregisterListener` at first asserts the permission (Line 3 in Figure 6.2) and then logs the config information (Line 14 in Figure 6.2). Thus, the placement of the permission check affects the information that could be leaked in these entry points as well. The `registerListener` revealing the car config information while `unregisterListener` not revealing anything unless permission is granted.

6.3.2 Case 2: `CarPowerManagementService`

The `CarPowerManagementService` in GM-Cad-Lyriq-SUV-24 (V12L) has two APIs: `finished` and `unregisterListener`, shown in Figure 6.3 and Figure 6.4. The `finished`

API (Figure 6.3) performs two checks before proceeding (*marked in red*): first, it ensures that the caller has the necessary permission `(Car. PERMISSION_CAR_POWER)` to interact with the car's power system, and second, it verifies that the method is being called from either a system process or the same process to ensure proper authorization. Once these checks are passed, the listener is converted to its Binder form and passed to the `finishedImpl` method, which is the primary sink of the API (*marked in green*), that handles the logic for completing the task and triggering the signal complete action. On the otherhand, the `unregisterListener` API (Figure 6.4) performs a single check to ensure that the caller has the required permission `(Car.PERMISSION_CAR_POWER)` to interact with the car's power system (*marked in red*). Once this permission is confirmed, it calls the `doUnregisterListener` method to remove the listener. In `doUnregisterListener`, the listener is unregistered from two lists: `mPowerManagerListeners` and `mPowerManagerListenersWithCompletion`. If the listener is unregistered in the second list `(found = true)`, it triggers `finishedImpl` - the primary sink (*marked in green*) to handle any final actions, such as marking the task as complete. It is clearly seen that in both the API calls, the common and primary sink is the `finishImpl` internal method (*marked in green*). But the access controls placed to reach to the sink is stronger in the foremost API than in the latter. Since the `(Car.PERMISSION_CAR_POWER)` permission is a signature level permission, the APIs can not be exploited to reach the sink. However, this is still an inconsistency that was caught by *AutoAcRaptor* and is indeed an anomaly. This case was reported to GM, who has acknowledged it and promised to address it in their next release.

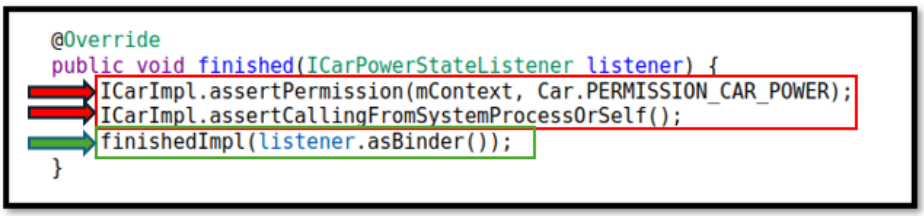

Figure 6.3: Implementation of `CarPowerManagementService. finished` retrieved from custom image, **GM-Cad-Lyriq-SUV-24 (V12L)**.

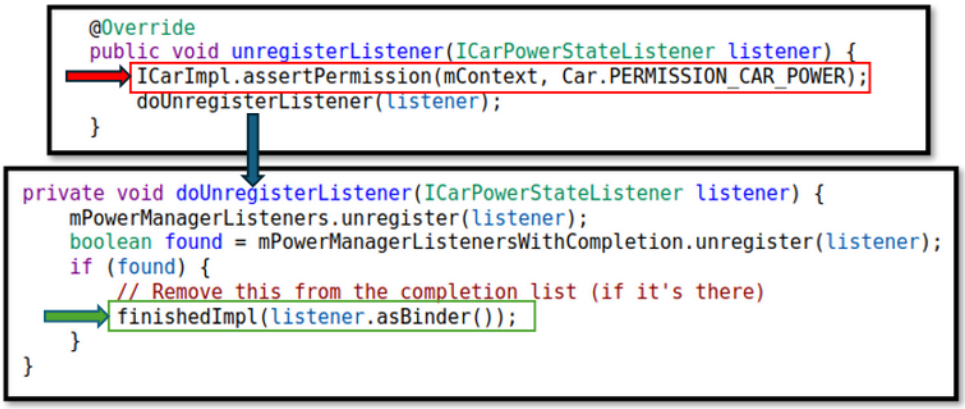

Figure 6.4: Implementation of `CarPowerManagementService. unregisterListener` retrieved from custom image, **GM-Cad-Lyriq-SUV-24 (V12L)**.

6.3.3 Case 3: `ConnectivityService`

Figure 6.5 shows `startNattKeepalive (API 1)` and `startNattKeepaliveWithFd (API 2)` - two APIs in the `ConnectivityService` class in the Volvo-XC40 (V11) image. Both APIs ultimately call the same primary sink, which is the `startNattKeepalive (1)` method in the `KeepaliveTracker` class. However, there are differences in how they reach this common sink. *API 1* requires a signature-level permission `(PACKET_KEEPALIVE_OFFLOAD)` before it can proceed. After the permission check, it invokes the `startNattKeepalive (1)` method in the `KeepaliveTracker` class. On the other hand, *API 2* calls a different version of the method, `startNattKeepalive (2)`. The `startNattKeepalive (2)` method then delegates the task to `startNattKeepalive (1)`, which means that both APIs ultimately call the same underlying method `(startNattKeepalive (1))`. The key distinction between these two APIs is that *API 1* is protected by a permission check, ensuring that only authorized processes can initiate the *keepalive* operation, while *API 2* lacks this permission check. This creates an inconsistency in access control between the two APIs, as both ultimately call the same internal method (taking different paths) but with different levels of security protection.

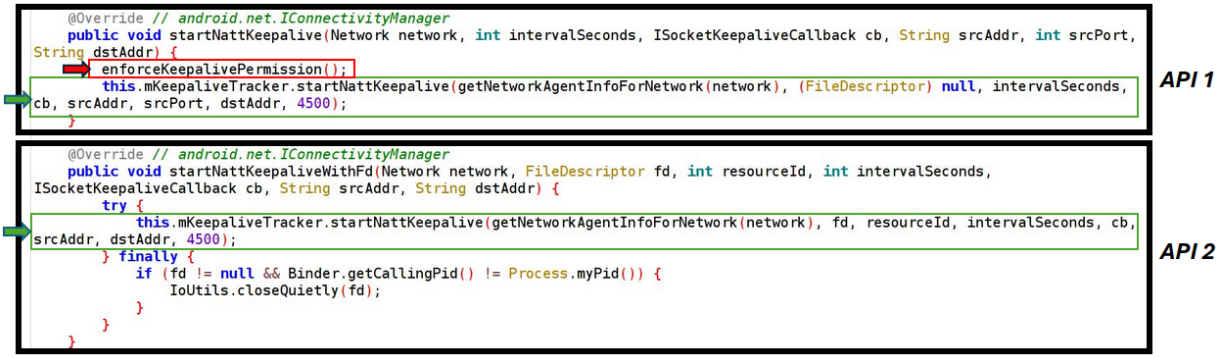

Figure 6.5: Simplified Implementations of `ConnectivityService. startNattKeepalive` and `startNattKeepaliveWithFd` retrieved from custom image, **Volvo-XC40 (V11)**.

Chapter 7

Related Works

Android Automotive Framework. Even though AAOS is a relatively new framework, its privacy and security has already been a focus of scrutiny by several recent works [54, 40, 51, 50]. The authors in [54] manually analyze 14 third party apps specifically built for AAOS and made available through the in-vehicle Google Play Store, and find that 78% of them have privacy policies that are inconsistent with the dangerous permissions they request. [51] surveys the privacy related attacks against AAOS such as driver fingerprinting, location inference, and driving behaviour analysis. PriDrive [40] uses network traffic inspection to derive the privacy landscape in AAOS apps by analyzing their data collection practices, whereas, Pricar [50] provides a privacy preserving data sharing framework for third-party apps. [44] paved the way by analyzing the security of in-vehicle infotainment systems and other similar app platforms. [52] narrows the focus by studying the overall architecture of AAOS specifically providing a classification of automotive attacks along with an analysis of existing security measures placed by Google to prevent such attacks. [25] surveys the different types of security vulnerabilities discovered in the AAOS framework. These existing works focus either on discovering specific vulnerabilities present in the AAOS, or manually analyze the architecture of the AAOS to clearly define the threat model and the attack surface. In contrast, we provide the *first systematic study* of the access control landscape of AAOS framework through an automated in-image and cross-image analysis powered by static analysis techniques; thereby discovering access control anomalies that may potentially result in serious security vulnerabilities.

Access Controls. Our work is closely related to existing approaches that analyze the traditional Android ROMs to discover access control issues [34, 17, 65, 21, 32, 22]. Stowaway [34] builds a permission specification for traditional Android ROMs using dynamic

analysis, whereas, PScout [21] improves these specifications using static analysis. Axplorer [22] improves upon these specifications by identifying and categorizing protected resources within Android. Arcade [20] performs path-sensitive analysis to construct precise protection maps for framework APIs. Another line of work [17, 65, 42, 32, 66] performs inconsistency analysis to identify different paths that lead to the same protected resource but enforce different access control. Kratos [65] compares different paths within Java layer to identify access control inconsistencies, whereas, AceDroid [17] improves upon Kratos by normalizing the access controls before comparing them. Poirot [32] introduces probabilistic analysis to account for the inherent uncertainty associated with identifying the access control specification for APIs. Bluebird [66] and FReD [42] compares paths leading to the same protected resource across different layers of the Android framework. *AutoAcRaptor* is closely related to these inconsistency detection approaches as it also performs in-image and cross-image comparison of access control specification for APIs. However, these existing tools are tailored specifically to work on traditional Android ROMs, and therefore, cannot achieve the same coverage as *AutoAcRaptor* when applied to AAOS ROMs as they cannot identify car-specific entrypoints.

Several other work study the security of other automotive platforms. [26] examines the external attack surface of modern cars, revealing vulnerabilities in diagnostic tools, media players, Bluetooth, and cellular networks that enable remote exploitation, control, and data exfiltration. It identifies systemic issues and offers recommendations for enhanced security. Similarly, [43] analyzes the security risks of smartphone integration with In-Vehicle-Infotainment (IVI) systems, specifically focusing on MirrorLink. The study uncovers vulnerabilities allowing attackers to exploit compromised smartphones and send malicious messages to the vehicle's internal network, advocating for improvements in security architecture. [33] studies attacks on the Control Area Network (CAN)-bus that is widely used in modern automotive platforms and discussed the open challenges that remain unanswered. VeCure [67] provides a practical security framework to protect against these CAN-bus related attacks, whereas, CAN-CID [59] provides a context-aware intrusion detection system that is based on a learning model that detects CAN-based fabrication and masquerading attacks in real time.

Chapter 8

Conclusion

AAOS has many extended and unique functionalities for automotive systems that allows OEMs to leverage them in their vehicles and customize as per their requirements. Despite being an extended version of the Android framework, unlike Android its access control logic has not been evaluated. We are the first to explore the framework in the access control implementation landscape. We propose an automated tool *AutoAcRaptor*, which is an enhanced version of traditional approaches that uses static analysis and lightweight NLP to generate and systematically evaluate access control specifications for AAOS framework entry points. We discuss our observations on the automotive-specific additions and modifications w.r.t Android OS to AAOS and to vendor customizations, with insights on the access control implementations depicted. Evaluating *AutoAcRaptor* on 10 AAOS ROMs revealed that customizations often lead to access control anomalies. As customizations increase, these anomalies are likely to become more prevalent in the future. Hence, our analysis highlights that automotive entry points are prone to access control anomalies, emphasizing the need for further scrutiny in AAOS frameworks.

References

[1] Android for Cars overview. https://developer.android.com/training/cars, 2024.

[2] Apktool. https://apktool.org/, 2024.

[3] CarServiceImpl.java. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/service/src/com/android/car/CarServiceImpl.java;l=82;drc=b4d6320e2ae398b36f0aaafb2ecd83609d2d99af, 2024.

[4] dex2jar. https://github.com/pxb1988/dex2jar, 2024.

[5] Git repositories on android. https://android.googlesource.com/, 2024.

[6] GM Developers. https://developer.gm.com/in-vehicle-apps, 2024.

[7] Honda Android Automotive OS Emulator. https://global.honda/en/cars-apps/index.html, 2024.

[8] ICarImpl.java. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/service/src/com/android/car/ICarImpl.java;l=441;drc=b4d6320e2ae398b36f0aaafb2ecd83609d2d99af, 2024.

[9] Polestar developer portal. https://www.polestar.com/global/developer#emulator, 2024.

[10] PropertyPermissionMapping.java. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/src/com/android/car/internal/PropertyPermissionMapping.java;l=25;bpv=0;bpt=1?q=PropertyPermiss&sq=&ss=android%2Fplatform%2Fsuperproject%2Fmain, 2024.

[11] ServiceManager.java. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/ServiceManager.java, 2024.

[12] smali. https://github.com/google/smali, 2024.

[13] Vehicle Properties. https://source.android.com/docs/automotive/vhal/previous/properties, 2024.

[14] VehiclePropertyIds. https://developer.android.com/reference/android/car/VehiclePropertyIds, 2024.

[15] VehicleVendorPermission. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/src/android/car/hardware/property/VehicleVendorPermission.java?q=VehicleVendorPermission, 2024.

[16] VOLVO: Android emulator. https://developer.volvocars.com/in-car-apps/android-emulator-xc40/, 2024.

[17] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[18] Y. Aafer, X. Zhang, and W. Du. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *Proceedings of the 25th USENIX Conference on Security, ser. SEC'16*, 2016.

[19] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1151–1164, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164, 2018.

[21] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, pages 217–228. Association for Computing Machinery, 2012.

[22] M. Backes, S. Bugiel, E. Derr, P. Mcdaniel, D. Octeau, and S. Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *USENIX Security Symposium*, 2016.

[23] Car.java. Car.java. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/src/android/car/Car.java, n.d. Accessed: 2023-12-22.

[24] CATI. Infographic: Learn the evolution of car infotainment systems. https://www.cati.ca/infographic-the-evolution-of-car-infotainment-systems/, July 2023. Accessed: 2023-12-20.

[25] Efstratios Chatzoglou, Georgios Kambourakis, and Vasileios Kouliaridis. A multi-tier security analysis of official car management apps for android. *Future Internet*, 13(3):58, 2021.

[26] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. https://www.usenix.org/conference/usenix-security-11/comprehensive-experimental-analyses-automotive-attack-surfaces, 2011. Accessed: 2023-12-20.

[27] cs.android.com. Car-lib. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/, n.d. Accessed: 2023-12-20.

[28] cs.android.com. Car.java. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/src/android/car/Car.java, n.d. Accessed: 2023-12-20.

[29] Volvo Cars Developer. Android Emulator XC40. https://developer.volvocars.com/in-car-apps/android-emulator-xc40/, n.d. Accessed: 2023-12-20.

[30] Android Developers. Test using the Android Automotive OS emulator. https://developer.android.com/training/cars/testing/emulator, n.d. Accessed: 2023-12-20.

[31] S. S. Dilip. Android Automotive: Shifting Gears from Mobile to Automotive. https://medium.com/@sreelakshmis.dilip/

android-automotive-shifting-gears-from-mobile-to-automotive-c2e4f7c8f15e, January 2024. Accessed: 2023-12-20.

[32] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically Recommending Protections for the Android Framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 937–950. Association for Computing Machinery, 2022.

[33] Faten Fakhfakh, Mohamed Tounsi, and Mohamed Mosbah. Cybersecurity attacks on CAN bus based vehicles: a review and open challenges. *Library hi tech*, 40(5):1179–1203, 2022.

[34] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[35] GitLab. Android Dumps · GitLab. https://dumps.tadiphone.dev/dumps, n.d. Accessed: 2023-12-20.

[36] Honda Global. Cars & Apps. https://global.honda/cars-apps/index.html, n.d. Accessed: 2023-12-20.

[37] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 25–36, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Bulut Gozubuyuk and Mert Pesé. PRIDRIVE: An Advanced Privacy Analysis Tool for Android Automotive. *Vehiclesec*, 2024.

[39] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[40] B. Gözübüyük, B. Tang, K.G. Shin, and M.D. Pesé. Analyzing Privacy Implications of Data Collection in Android Automotive OS. *arXiv.org*, September 2024.

[41] Steve Corrigan HPL. Introduction to the controller area network (CAN). *Application Report SLOA101*, pages 1–17, 2002.

[42] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. FReD: Identifying File Re-Delegation in Android System Services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, Boston, MA, August 2022. USENIX Association.

[43] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A Security Analysis of an In-Vehicle Infotainment and App Platform. https://www.usenix.org/conference/woot16/workshop-program/presentation/mazloom, 2016. Accessed: 2023-12-20.

[44] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. A security analysis of an {in-vehicle} Infotainment and App platform. In *10th USENIX workshop on offensive technologies (WOOT 16)*, 2016.

[45] General Motors. In-Vehicle Apps Development. https://developer.gm.com/in-vehicle-apps, n.d. Accessed: 2023-12-20.

[46] M. Najafi, M. Lemercier, and L. Khoukhi. Data leakage prevention model for vehicular networks. In *2022 18th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 124–129, 2022.

[47] A. Oyler. Differentiating Android Automotive AOSP and Google services. https://www.linkedin.com/pulse/differentiating-android-automotive-aosp-google-services-alex-oyler, n.d. Accessed: 2023-12-20.

[48] Amit Ahlawat Hao Hao Yifei Wang Paul Ratazzi, Yousra Aafer and Wenliang Du. A Systematic Security Evaluation of Android's Multi-User Framework. In *arXiv*, 2014.

[49] A. Pawar. Exploring Android Automotive Car Service and Third-Party App Development. https://medium.com/softaai-blogs/exploring-android-automotive-car-service-and-third-party-app-development-d71df6355dfc, August 2024. Accessed: 2023-12-20.

[50] Mert D Pesé, Jay W Schauer, Murali Mohan, Cassandra Joseph, Kang G Shin, and John Moore. PRICAR: Privacy Framework for Vehicular Data Sharing with Third Parties. In *2023 IEEE Secure Development Conference (SecDev)*, pages 184–195. IEEE, 2023.

[51] Mert D Pesé and Kang G Shin. Survey of automotive privacy regulations and privacy-related attacks. 2019.

[52] M. Pesé, K. Shin, J. Bruner, and A. Chu. Security Analysis of Android Automotive. *SAE Technical Paper*, 2020-01-1295, 2020.

[53] M. Pesé, Kang Shin, Josiah Bruner, and Amy Chu. Security Analysis of Android Automotive. *SAE International Journal of Advances and Current Practices in Mobility*, 2, 2020.

[54] M.D. Pesé. A First Look at Android Automotive Privacy. *SAE Technical Paper*, 2023-01-0037, 2023.

[55] Polestar. Developer - Get Started - Emulator. https://www.polestar.com/us/developer/get-started/emulator/, n.d. Accessed: 2023-12-20.

[56] Android Open Source Project. Android Base Frameworks Source Code. https://android.googlesource.com/platform/frameworks/base/+/refs, n.d. Accessed: 2023-12-20.

[57] Android Open Source Project. Security updates and resources. https://source.android.com/security/overview/updates-resources#severity, n.d. Accessed: 2023-12-20.

[58] Android Open Source Project. What is Android Automotive? https://source.android.com/docs/automotive/start/what-automotive, n.d. Accessed: 2023-12-20.

[59] Sampath Rajapaksha, Harsha Kalutarage, M Omar Al-Kadri, Garikayi Madzudzo, and Andrei V Petrovski. Keep the moving vehicle secure: Context-aware intrusion detection system for in-vehicle CAN bus security. In *2022 14th International Conference on Cyber Conflict: Keep Moving!(CyCon)*, volume 700, pages 309–330. IEEE, 2022.

[60] WALA GitHub Repository. WALA GitHub Repository. https://github.com/wala/WALA, n.d. Accessed: 2023-12-20.

[61] SamMobile. Home - SamMobile. https://www.sammobile.com/, June 2022. Accessed: 2023-12-20.

[62] SamMobile. Porsche cars to feature Android Automotive, Google Play Store integration soon. https://www.sammobile.com/news/porsche-cars-android-automotive-google-play-store-soon/, 2023. Accessed: 2023-12-20.

[63] Android Code Search. CarLocalServices. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/service/src/com/android/car/CarLocalServices.java;drc=b4d6320e2ae398b36f0aaafb2ecd83609d2d99af;bpv=0;bpt=1;l=65, n.d. Accessed: 2024-12-19.

[64] Android Code Search. CarService. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/service-builtin/src/com/android/car/CarService.java?q=CarService&ss=android%2Fplatform%2Fsuperproject%2Fmain, n.d. Accessed: 2023-12-19.

[65] Yuru Shao, Qi Alfred Chen, Z. Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Network and Distributed System Security Symposium*, 2016.

[66] Parjanya Vyas, Asim Waheed, Yousra Aafer, and N. Asokan. Auditing Framework APIs via Inferred App-side Security Specifications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6061–6077, Anaheim, CA, August 2023. USENIX Association.

[67] Qiyan Wang and Sanjay Sawhney. VeCure: A practical security framework to protect the CAN bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)*, pages 13–18. IEEE, 2014.

[68] X. Zhang, Y. Aafer, K. Ying, and W. Du. Hey, You, Get Off of My Image: Detecting Data Residue in Android Images. In *European Symposium on Research in Computer Security*, 2016.