Android Access Control Recommendation as a Deep Learning Task

by

Dheeraj Vagavolu

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, 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Statement of Contributions

This thesis consists of material which I co-authored with my supervisors, Professor Yousra Aafer and Professor Meiyappan Nagappan, which has been submitted to ICSE 2024 conference and is currently under review.

Abstract

Android enforces access control checks to protect sensitive framework APIs. If not properly protected, framework APIs can open the door for malicious apps to access sensitive resources without having the necessary privileges. Unfortunately, as reported in the existing literature, such access control anomalies are prevalent in Android APIs, notably those introduced by customization parties. Therefore, various solutions have been proposed to detect anomalies, particularly those due to inconsistencies in the enforcement of access checks across the Android framework(s). The solutions can be largely divided into two categories: convergence-based techniques which rely on the convergence of two APIs on similar resources, and probabilistic approaches which incorporate additional hints in the form of manually defined structural and semantic code constructs. In this paper, we are motivated by the promising application of using code constructs, beyond convergence as proposed by the probabilistic approaches, to recommend access control enforcement and detect inconsistencies.

Specifically, we propose a deep learning-based approach that aims to automatically learn the correspondence between various code constructs and access control requirements. To this end, we fine-tune CodeBert on statically derived features from the Android Open Source Project (AOSP). Our feature engineering process addresses various peculiarities that characterize Android implementations. The resulting fine-tuned model can be queried to recommend access control for vendor-customized APIs.

The fine-tuned model achieves an accuracy of 93%, a precision of 91%, and a recall of 92% in the AOSP data. Additionally, our evaluation of custom ROMs shows that the model is able to not only rediscover previously reported inconsistencies but also discover new ones.

Acknowledgements

I would like to thank my supervisors, professor Mei Nagappan and professor Yousra Aafer for guiding me during my Master's. My appreciation also goes to Zeinab El-Rewini and Parjanya Vyas for their invaluable assistance.

Dedication

This work is dedicated to my Parents, Brother, Friends, and Alizeh, in recognition of their unwavering support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android Framework APIs enforce access control checks to prevent unauthorized access from third-party applications. Erroneous enforcement of access control checks can result in potential vulnerabilities that put mobile users at risk. Due to the lack of security specifications for Android APIs, the research community has adopted inconsistency analysis to detect underprotected APIs; which can be divided into two main strategies: convergence analysis and probabilistic inference. Kratos [27], AceDroid [3], and ACMiner [16] are examples of convergence analysis-based approaches. They assume that system service APIs that converge on the same resource must have similar access control policies. For example, if two APIs allow deleting the same file (the delete operation is the convergence point) such that two APIs enforce different access control, then an inconsistency is detected. The API enforcing the weakest access control is flagged as likely vulnerable. However, these tools tend to generate false positives due to the imprecise nature of the assumption that powers these tools: *the convergence point might not be security sensitive*. For example, consider the case where two APIs converge on an insensitive database sync operation.

Recently, probabilistic approaches have been proposed to address the inherent limitation of convergence analysis. Poirot [10] regards the relationship connecting access control to reachable instructions as *uncertain*. As such, it models the intuition that a reachable resource might not be the target of a preceding access control. Poirot solves the uncertainty by collecting various *probabilistic rules* – i.e., structural and semantic code constructs and properties connecting access control to resources and resources to each other. Probablistic inference is then employed to aggregate the constructs and properties in order to project a protection recommendation for a resource.

Although promising, Poirot suffers from a few limitations: (1) the probabilistic rules

are manually compiled using domain knowledge; as such, they might not comprehensively capture all diverse properties in Android APIs. (2) Due to the specific nature of the rules, certain resources might not exhibit sufficient observations, and hence Poirot cannot project a high-confidence access control recommendation for these resources.

To address the limitations of existing work, we advocate for a deep learning-based approach which can automatically learn the correspondence between *structural and semantic features* of Android APIs and an *access control label*. Unlike probabilistic inference tools, automatic learning of *rules* can allow our approach to have a more complete overview of various code constructs that warrant access control. Recent studies have shown that deep learning-based approaches can achieve state-of-the-art performance in various software engineering (SE) tasks of similar nature to ours, such as vulnerability detection and malware detection [11, 19, 26]. However, adapting a deep learning-based approach to recommend accurate access control poses several challenges. First, Android APIs may enforce conjoined/disjoined access control depending on the supplied arguments and system properties. Therefore, it is challenging to derive a *label* at the level of the whole API. Second, the path-sensitive nature of APIs implementations implies that each execution path may reflect different code properties; as such, two execution paths may require different labels. Therefore, expecting a deep learning model to learn path-specific correspondences from the entire API implementation may not be suitable. These challenges motivated us to propose a new API representation that abstracts the implementation to execution paths. Specifically, we detect individual execution paths leading to various resources and extract the respective access control enforced along the path. The resulting features provide a more fine-grained and precise representation of access control targets compared to using all the instructions in an API.

Our proposed method addresses various technical challenges observed while curating the features. Essentially, we introduce a few optimizations and insights to guide the model. Among others, we statically preprocess and reduce the paths to eliminate prevalent noisy patterns (seemingly protected code constructs that are actually insensitive). We perform IR decompilation to convert WALA IR instructions into a form that is amenable to the model. We further decompose paths into subsequences to allow the model to learn the cases where only a subset of the path instructions is the target of access control. To derive the labels accurately, we perform path-sensitive access control extraction and normalization. The evaluation of our approach yields an accuracy of 93%, a precision of 91%, and a recall of 92% in the test data set. We further evaluate the effectiveness of our tool in identifying security sensitive access control inconsistencies in vendor-customized ROMs by manually verifying the APIs flagged by our tool. Our findings demonstrate that our approach is capable of identifying both already known and new cases of access control inconsistencies.

Chapter 2

Background & Motivation

Detecting access control anomalies in Android APIs has long been a center of attention by the Android research community for various reasons. First, the security specification for Android APIs is known to be incomplete for public APIs and missing for private APIs. Second, due to the large code base size of the Android framework and the highly diverse and complex nature of Android access control, it is challenging for Android OEMs to ensure that access and operations on all sensitive resources are adequately protected in a consistent manner. Third, such inconsistencies imply that attackers with insufficient privileges can access the under-protected resource, thus leading to various privacy and security consequences. And fourth, the Android framework allows OEMs and device manufacturers to incorporate significant customizations to the system services. Such customizations often introduce new end points, increasing the risk of access control inconsistencies if proper checks are not enforced.

To address these issues, the research community has developed methods that can detect inconsistencies in access control enforcement. In general, the approaches work either through convergence analysis [3, 16, 17, 27, 34] or through probabilistic inference [10]. Although the two approaches have been successful in finding substantial access control anomalies, they suffer from some limitations. Next, we describe the general methodology of these approaches and describe the relevant limitations, which have motivated our work.

2.1 Convergence-based Inconsistency Detection

.

This approach asserts that two or more system service APIs that converge on the same resource must require a similar level of protection. The earliest effort, Kratos [27] detects inconsistencies by looking at the access control checks along the paths leading to the same sensitive resources. AceDroid [3] improves Kratos's method by addressing the diverse nature of Android access control, which was mainly responsible for Kratos's false positives. Specifically, AceDroid normalizes and converts diverse access control checks along the paths leading to same resource into a canonical representation. As such, syntactically diverse but semantically similar checks will be mapped to the same form, hence reducing false positives. ACMiner [16] uses text analysis techniques to statically detect potential authorization checks, which may be overlooked by Kratos and AceDroid. As such, ACMiner can detect exclusive access control vulnerabilities. IAceFinder [34] extends convergence-based approaches to pinpoint inconsistent access control enforcement across the Java context and native context of Android. FReD [17] identifies another type of cross-context access control inconsistency; those where Java APIs APIs accessing a file and the actual concrete file path are protected differently.

2.1.1 Limitations of Convergence Analysis

. Despite their highly promising application in detecting access control anomalies, convergence based methods suffer from the following significant inaccuracies.

- (**LC-1**): First, the methods over-approximate access control requirements for resources as they rely solely on control dependencies, i.e., a given resource is assumed to be protected by an access control check if it is control dependent on it. However, this may be inaccurate if the check is targeting other sensitive resources co-located with the given resource. Therefore, the assumption could inherently lead to false positives.

- (**LC-2**): Existing convergence analysis methods can only identify explicit *reachability-based* inconsistencies. That is, a vulnerability can be detected if two APIs, enforcing two distinct access control requirements, lead to the same resource. However, resources in Android may be connected via other types of implicit relations. Therefore, the existing approaches will fail to detect pertaining vulnerabilities.

In addition, to understand the limitations described in **LC-2**, consider the following system service API clearAutoHotspotLists() in the class SemWifiServiceImpl shown in Figure 2.1.

Figure 2.1: Call chain of clearAutoHotspotLists() Method in SemWifiServiceImpl Class

This method calls another method with the same name in the internal class SemWifiApServiceImpl, which invokes the method clearLocalResults() in SemWifiApSmartClient. The method clearLocalResults operates on the private fields *mWifiApBleScanResults* and *mSmartMHSDevices*, completely removing their contents. The *mWifiApBleScanResults* and *mSmartMHSDevices* fields play a significant role within the SemWifiApSmartClient class. These fields are accessed and used in multiple places throughout the class. Their repeated usage suggests that they contain essential information critical to the functionality of the SemWifiApSmartClient. Considering the sensitive nature of hotspot-related operations, protecting the data stored in these fields is crucial. Unauthorized access to completely clear the data stored in the *mWifiApBleScanResults* and *mSmartMHSDevices* fields could lead to disturbances in the proper functioning of SemWifiApSmartClient.

Upon inspection, we observe that convergence to the reachable resources clearLocalResults(), mWifiApBleScanResults, and mSmartMHSDevices do not exist across the Android ROM. Therefore, the missing access control in the system service API clearAutoHotspotLists cannot be detected by convergence analysis-based approaches, demonstrating the limitation described in **LC-2**.

To address the lack of convergence in the system service API, we can consider inspecting a method which is similar in structure and implements similar functionality in a different context. For example, consider the system service API clearDebuggingKeys() in ADBManagerService which clears debugging keys and enforces the calling or self permission of "android.permission.MANAGE_DEBUGGING". This access control measure ensures that only authorized callers can perform debugging-related operations. Similar to clearAutoHotspotLists(), clearDebuggingKeys deletes the content of the private fields, mConnectedKeys and mWifiConnectedKeys, in the AdbDebugginManager class. The similarities in structure and semantics between the two APIs, clearDebuggingKeys() and clearAutoHotspotLists(), can be leveraged to provide motivation for the existence of the missing access control in the latter. By analyzing the access control measures and permission enforcement in clearDebuggingKeys(), we can infer with some confidence that a similar level of access control protection might be necessary for clearAutoHotspotLists().

Although convergence analysis-based approaches will not be able to detect the missing access control for the method clearAutoHotspotLists(), we can draw insights from similar methods, such as clearDebuggingKeys(), to anticipate the need for access control in clearAutoHotspotLists() based on shared structural and semantic characteristics. To this end, we aim to provide a solution that can learn such structural and semantic patterns in system service APIs to predict the requirement of access control enforcement, addressing the limitation **LC-2**.

Figure 2.2: Call chain of clearDebuggingKeys() Method in AdbService class

2.2 Probabilistic Inconsistency Detection

To tackle the inaccuracies of existing convergence-based approaches, Poirot [10] reconceptualizes inconsistency detection to account for uncertainty. It follows a different angle to pinpoint inconsistencies. It leverages a set of rules (derived from Android domain knowledge) to *probabilistically infer* an access control recommendation for resources and APIs. The recommendations can naturally be used to detect access control inconsistencies – i.e., an inferred recommendation is different from the actual implementation.

Technically, Poirot works as follows: it begins by extracting a broad range of probabilistic relations that (1) assign access control to resources and (2) link resources to each other, hence allowing to transitively transfer access control across linked resources. The relations encapsulate both certain and uncertain code constructs, termed *facts* and *hints*, respectively. An example of a fact is 1-1 control dependencies (i.e., one resource is protected by a single access control check). Hints are less certain than facts; for instance, if two resources *r1* and *r2* share similar names, we can speculate that *r1*'s protection is required for *r2*. Lastly, Poirot uses a probabilistic inference engine to aggregate the facts and hints to project a high confidence protection recommendation for a resource.

2.2.1 Limitations of Probabilistic Analysis

. Although Poirot proved to be successful in detecting new vulnerabilities and suppressing false alarms of convergence-based approaches, it suffers from the following limitations.

- **(LP-1)**: Poirot depends on a set of rules (i.e., code constructs and program properties), that are *manually compiled and derived through domain knowledge*. As such, their completeness is not guaranteed.
- **(LP-2)**: Poirot models the importance of the rules by using preset probability values. Although probabilistic inference is insensitive to variations in these values [10], we note that a wrongly present value may affect the recommendation accuracy; consider an uncertain clue that Poirot deemed to be highly certain.
- **(LP-3)**: The rules are highly specific to a target resource(s) and cannot be applicable to other resources that share semantic properties with the target.
- **(LP-4)**: Third, due to the rules' narrow application (or specificity), certain resources may not exhibit a sufficient number of rules, that is enough for Poirot to learn from.

As reported, a lower number of observations (pertaining to the rules) can hinder the ability of the probabilistic engine to make confident recommendations. This is indeed demonstrated by Poirot's (relatively) low coverage.

Using the system service class *AmazonInputManagerService* in the Amazon Fire HD 10 device, we demonstrate the limitation of manually defining the rules and probability values **(LP-1)** and **(LP-2)**, of the current probabilistic approach and illustrate the effectiveness of our proposed solution to address these limitations. We provide the simplified implementation of the methods **(A)** *getInputLockingMode()*, **(B)** *setInputLockingMode(int mode)*, and **(C)** *showInputLockingMode(int inputLockingMode)* in Figures 2.3, 2.4, and 2.6 respectively.

```
1  public int getInputLockingMode() {
2      checkPermission(AmazonInputManagerService.PERMISSION_INPUT_LOCK);
3      return AmazonInputManagerService.this.mInputLockingMode;
4  }
```

Figure 2.3: **(A)** Strong Clue that **mInputLockingMode** requires Signature Permission

```
1  public boolean setInputLockingMode(int mode) {
2      checkPermission(AmazonInputManagerService.PERMISSION_INPUT_LOCK);
3      if (mode == 0 || mode == 1) {
4          AmazonInputManagerService.this.mInputLockingMode = mode;
5          ...
6          MetricsHelper.recordInputLockingModeUpdateEvent(mode);   (R1)
7          showInputLockingNotification(mode, true);                (R2)
8          return true;
9      }
10     ...
11     return false;
12 }
```

Figure 2.4: **(B)** Low confidence propagation of protection for the resource *showInputLockingNotification(mode, true)* due to manually tuned control rules in Poirot

First, by performing a depth-first search (DFS), Poirot [10] would be able to detect that the field this.mInputLockingMode has a one-to-one relation with the signature level permission PERMISSION_INPUT_LOCK and assign it a high confidence score. Hence, a high confidence recommendation for AmazonInputmanagerService.this.mInputLockingMode can be determined in method A (Fig. 2.4). Further, Poirot uses rules to control the propagation of access control checks for resources using implicit rules such as parameter flow, naming similarity, and one-to-many mapping using manually set confidence scores. Using probabilistic inference, Poirot aggregates all the probabilities and generates final predictions based on a selected cutoff. However, we observe that using rules with manually tuned probabilities, Poirot cannot determine with high confidence that the resource *showInputLockingNotification(inputLockingMode, false)* (R2) in the method (B) requires an access control protection. Thus, Poirot cannot detect the missing access control check for R2 in method (C).

```
1   public void showInputLockingNotification(
2               int inputLockingMode, boolean cancelPrevNotif) {
3
4       long identity = Binder.clearCallingIdentity();
5
6       ... (debug features)
7       ... (new NotificationChannel)
8       ... (new Notification.Builder)
9       ... (set notification and builder properties)
10
11      AmazonInputManagerService.this.
12                          mNotificationManager.
13                          notify(
14                              NOTIFICATION_TAG, 1, builder.build()
15                          );
16
17      Binder.restoreCallingIdentity(identity);
18  }
```

Figure 2.5: Implementation of the resource showInputLockingNotification (R2)

```
1   public void showInputLockingNotification(int inputLockingMode) {
        // Missing Access Check
1
2       showInputLockingNotification(inputLockingMode, false);  (R2)
3   }
```

Figure 2.6: **(C)** Missing Access Check in method *showInputLockingNotification*

Lowering the cutoff percentage to include this case as a potential inconsistency could cause several other cases to be included resulting in an increased number of False Positives. For example, the resource *recordInputLockingModeUpdateEvent(mode)* (R1) in method (B) also has implicit one-to-many relation, naming similarity, and a parameter flow from the caller. Hence, similar probability scores will be propagated through the probabilistic inference. If the cutoff percentage were any lower, Poirot would flag R1 and R2 as requiring Access Control Protection. However, upon closer inspection of the implementation, we see that R1 is used for logging and updating fields internally by the MetricsHelper class and does not require access control protection. From the implementation of R2 in Fig. 2.5, we see that this method can generate system-level notifications using the NotificationManager framework channel and does indeed require access control protection. However, Poirot could not distinguish between the access control requirements between the two resources, R1 and R2, without introducing addition rules, highlighting **LP-1**. Moreover, it could not take advantage of the implementation of the resource at a deeper level to recommend access control requirement for the resource based on other system service APIs with similar functionality, demonstrating the limitation **LP-2**.

2.3 Our Proposed Solution

To address the limitations discussed in Section 2, we propose a deep learning-based solution to automate the extraction of structural and semantic clues from the source code of framework APIs. Specifically, we leverage the capabilities of deep learning, through the use of a pretrained language model, CodeBert. We start by extracting individual execution paths in the API leading to sensitive resources and extracting the respective access control enforced along these paths. This process results in precise and granular data, compared to using instructions from the entire API, that address the limitation LC-1. Additionally, CodeBert is capable of automatically extracting the structural and semantic code features

from the source code, which is best suited for our task. Using these features, CodeBert can make predictions for individual execution paths without the need for detecting convergence, which addresses the limitation LC-2. Furthermore, automatic extraction of rules alleviates the need for human input in devising *clues* for access control recommendation, effectively addressing LP-1 and LP-2. By looking for rules over the entire dataset, the model learns generalizable patterns, which allow it to make predictions for semantically similar resources addressing the limitations LP-3 and LP-4.

Chapter 3

Priming for Deep Learning

In this section, we answer fundamental questions to guide the design of our proposed deep learning-based solution.

3.1 How to construct meaningful features to reflect access control enforcement in Android APIs?

Since deep learning tasks operate on vectors, we need to represent Android APIs implementations as features which can be converted to vectors by the appropriate models. In other words, we need to convert a given API implementation into features that (1) are the expected input for a specific deep learning model and, more importantly, (2) reflect precisely code patterns requiring access control. In literature BERT based language models have been extensively used for source code related tasks which expect a continuous piece of text as the input feature. Note that we cannot arbitrarily feed the whole API code due to the inherent complexities that characterize Android access control implementations.

Characteristic I: Android APIs can enforce conjoined and / or disjoint conditional access control. As such, the access control requirement (i.e., a *label* in our context) cannot be easily derived at the granularity of the whole API. As reported by Arcade [4], an API's access control can be represented as a first-order logic formula that captures security conditions and their correlations, which is inherently difficult to abstract as a single label.

Characteristic II: As noted in the Motivation Section 2, not all code blocks in an API are of interest to access control. For example, failed input validation logic and failed security

checks are unrelated to the API's functionality. Blindly assigning the API's label(s) to these *noisy* code may negatively impact the model's accuracy as reported in our experiment (see 5.3). The pre-trained model should be able to recognize that such blocks of code are not related to access control, given enough training data. However, due to the lack of ground truth data in the context of Android access control, we use static analysis to remove code that does not contribute to the API's functionality.

The above characteristics lead us to propose a new API representation that abstracts the implementation to *execution paths*. A path is a set of sequential instructions that (1) starts at the API's entry point, (2) can be tagged with a single access control label, and (3) should ideally capture semantic information pertaining to the label, i.e., contains the access control targets.

Note that the last requirement is challenging; as mentioned earlier, accurately locating access control targets in Android APIs is a highly uncertain process. It requires understanding various semantic properties (e.g., similarity between API and reachable instructions) and structural constructs (e.g., instructions must follow a specific order). As such, expecting the model to learn the correspondence between the target instruction(s) and the access control label is difficult, particularly in light of insufficient training data.

As we will elaborate later, this observation suggests that offering some guidance to the model may be helpful in learning the correspondences effectively. Hence, instead of feeding all instructions in a given execution path to the model as input (and expecting the model to infer the correspondence from such a sequence), we decompose the path into subsequences. Essentially, we transfer to the model our understanding that any subsequence of the execution could be the target of access control. More details on subsequences are discussed in Section 4.2.

3.2 Why Finetuning CodeBert?

Given the success of Poirot [10], it is evident that the structural constructs and semantic features of the Android APIs (termed *clues*) are crucial to recommend access control. Therefore, we sought a solution capable of understanding and automatically extracting such clues. Deep learning models have been very successful in inferring important features to solve complex software engineering tasks such as vulnerability [30, 7, 13, 22] and malware detection [15, 26, 23, 31, 5], all without the involvement of human experts.

Specifically, in line with the existing Android literature, we treat the Android Open Source Project (AOSP) code bases as ground truth, from which we statically construct

and automatically label a large corpus of training data. The statically constructed training corpus can be used to train a DL model, which can be queried to recommend access control (and detect anomalies) in vendor-customized ROMs.

By analyzing 3 AOSP ROMs, we observe that the constructed training set is relatively smaller with 20145 data points (73.9% 'Requires Protection' and 26% 'No Protection') compared to the 2.1M datapoints used to pre-train CodeBert [12]. Given the immense capacity of deep learning models for learning, training from scratch on such a relatively smaller and unbalanced data set is insufficient, potentially leading to overfitting and lack of generalizability [33, 18]. According to recent studies [19, 29], we have chosen to fine-tune CodeBert, a transformer pre-trained model that works with bimodal data (source code and documents) and has demonstrated promising results. As such, CodeBert is able to extract structural and semantic code constructs to make accurate recommendations.

3.3 What is the access control label/recommendation granularity?

As we represent APIs at a finer granularity, we need to pinpoint the exact access control requirement for each execution path leading to a given resource.

This implies that our training data labeling process requires *a path-sensitive analysis* of the APIs. In particular, for each execution path leading to a given resource, we identify access control checks (e.g., conditional statement enforcing a permission). The checks can then be conjointed (using `AND` operator) to represent the *label* for the resource-specific execution path. Given the diverse nature of Android permissions, we observe that such a process would lead to a significant number of labels. Therefore, we opt to use the normalization procedure proposed by AceDroid [3] to convert the highly diverse labels to a canonical format (leading to four labels). We further leverage the representation and our domain knowledge to reduce the number of labels to two labels due to class imbalance (more details are demonstrated in Section 4.5). This solution is sufficient for the present study, but may need to be refined if the goal is to propose a finer-grained access control recommendation.

Chapter 4

Approach

Figure 4.1 presents a high-level overview of our proposed solution. It consists of five phases: data collection, feature selection, reduction, IR decompilation, and model fine-tuning. As input, the solution expects (1) a set of Android ROMs for training, and (2) a target custom ROM to be analyzed for access control recommendation (and vulnerability detection). As output, the solution generates a mapping of custom APIs to access control recommendation.

4.1 Phase-1: Data collection and Pre-Processing

AOSP is known to have significantly less access control anomalies compared to vendor-customized ROMs [27, 3]. Therefore, we treat AOSP ROMS (versions 11, 12, and 13) as ground truth and use it for fine-tuning CodeBert. Observe that this choice is inline with the body of literature on Android access control evaluation. We preprocess the ROMs to collect important information for further phases, as follows.

We statically analyze the ROMs to identify app-accessible system services by looking for corresponding registration points. We look for the methods `addService` and `publishBinderService` which allow to publish public system services in the Service Manager[1]. We then resolve the registered system service class types. Afterwards, we extract the exposed interface class of the service and retrieve its declared public methods – these methods correspond to Android service APIs.

[1]Central registry that contains running system services

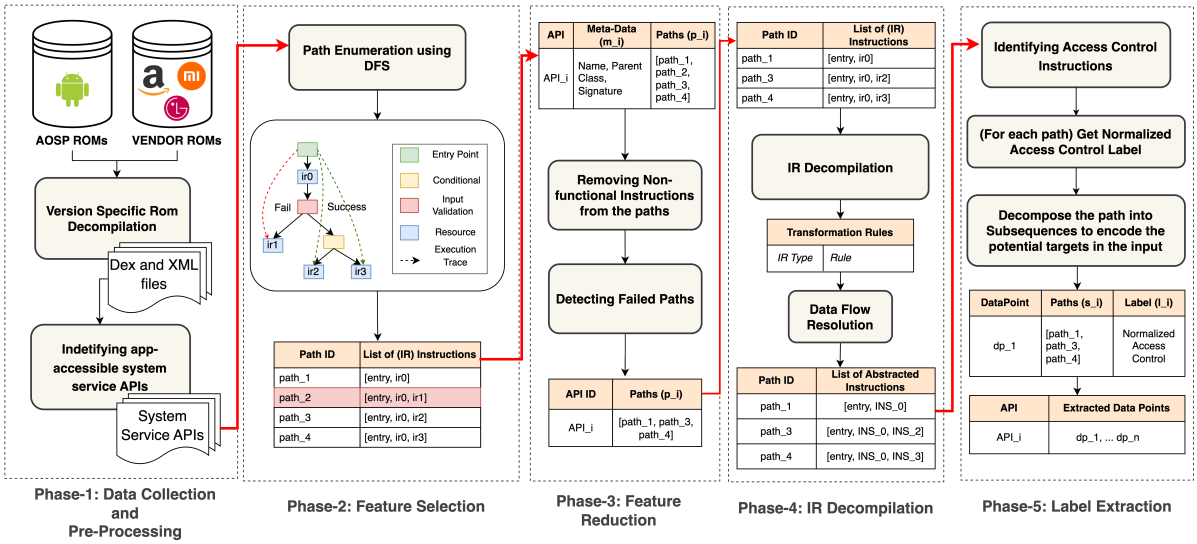

Figure 4.1: Data Collection Pipeline for Training CodeBert to Predict Access Control Inconsistencies

4.2 Phase-2: Feature Selection

Our solution enumerates all execution paths in each API. Formally, a path is composed of a number of sequential program statements which are correlated via control or data dependency. To extract the paths, we begin by building an interprocedural control flow graph (icfg) of each API. Starting from the entry basic block of the icfg, we perform a depth-first traversal along the control dependence edges to traverse all paths. The traversal ensures control dependencies among the statements are correctly preserved.

Optimization. Enumerating all the paths in the APIs requires significant resources. Hence, we make the following optimizations for the path traversal:

- **Fixed Depth of Traversal:** We consider a three-level depth for the DFS traversal of the icfg, which significantly reduces the total number of paths being traversed – though this can be configured based on the available resources.
- **Early Stopping during Access Control Enumeration:** During enumeration of Access Control Checks we stop early when we detect a higher level of access control (corresponding to 'dangerous' or 'system'). The Normalization process explained in Section 4.5 removes the need for further traversal.

- **Parallelization and Memoization:** We use parallel processing to enumerate paths for multiple APIs concurrently, thereby reducing the overall computation time. Further, we store the results of expensive or frequently-calculated path traversals for specific methods, so they do not have to be recalculated during the interprocedural traversal.

- **Early stopping of traversal in noisy paths:** We remove conditional branches associated with failed input validation and access control validation as described in Section 4.3, which allows us to stop the interprocedural DFS for methods in those removed parts.

Decomposing execution paths into subsequences. As noted earlier, inferring the precise correspondence between instructions (along a single path) and the access control label is a highly challenging task. Here, we aid the model to address this challenge by transferring our domain-knowledge that any (ordered) subsequence of instructions might be the target of access control.

Specifically, we decompose each identified execution path (which starts a API's entry point and ends at an exit node, e.g., return statement) into subsequences. We use a sliding window over the execution path. The window begins with a fixed initial size of one instruction starting from the first instruction that is control-dependent on an access control check. It then moves forward incrementally by d instructions, partially overlapping with the previous subsequence. The variable d denotes the number of instructions required to reach a *protected resource*; that is, a potentially sensitive instruction. We use the taxonomy provided by Axplorer [6] to identify *protected resources*, which can be categorized into method calls, field updates, method invocations, return (value) statements, and throw (*RuntimeException*) instructions. This choice ensures that a subsequence includes at least one potential protected resource.

The process is repeated until the sliding window covers the entire execution path. The resulting subsequences naturally overlap and depict a shorter execution of the decomposed execution path.

Generated features. This phase synthesizes the following pair for each Android API: *(m_i , p_i)* where m is metadata pertaining to the API (i.e., name, parent class, and signature), and p is the list of identified and execution paths. Each execution path *p_j* is further represented as a pair *(s_j, l_j)* where s is a list of a (decomposed) path's subsequences and l is the label for the path *p_j*. Observe that all subsequences of a particular execution path share the same label, implying that we aggregate all subsequences into a single feature.

In other words, we combine all subsquences that are protected by the same access control into a single feature.

Modifying CodeBert Architecture to incorporate multiple subsequences. Bert-based models rely on three main embeddings to generate the final embeddings for a given input sequence: token_embeddings, positional_embedding, and segment_embedding [9]. Token_embedding capture the uniqueness of each token, while positional_embedding captures the position of the token in relation to the other tokens in the input. In downstream tasks that require the input to be composed of multiple segments (such as question-answering and sentence-completion tasks), Bert-based models use special segment_embeddings to represent the two different segments in a single input. The models based on Bert architecture are designed to process a maximum of two segments in the input, which are separated by the *[SEP]* token. However, in our case, it is possible that the combined input contains more than two segments (subsequences). To ensure that the CodeBert tokenizer accurately reflects the separation of these segments, we have modified its architecture. Specifically, in addition to the *[CLS]* (special Start token) and *[SEP]* (special end token), we incorporate a special token *[SUB]* into the model that separates the 'n' segments (subsequences) in the combined input. Furthermore, we modify the tokenizer to use a distinct segment_embedding for each unique subsequence identified using the special token. Similarly, the truncation process is modified to give higher priority to longer subsequences since they retain the entire execution path. Upon fine-tuning, the model learns to use the special segment_embedding to make accurate recommendations.

4.3 Phase-3: Feature Reduction

Not all operations provided by an API are of interest to enforced access control; certain program statements or even a whole execution path may be unrelated to a detected access control check (along the path). Consider the simplified code snippet depicted in Listing 4.1, extracted from the public AOSP API `PackageManger. addCrossProfileIntentFilter`:

```
1 public void addCrossProfileIntentFilter(IntentFilter
     intentFilter, String ownerPackage, int sourceUserId, int
     targetUserId, int flags) {
2     ...
3     enforceCallingOrSelfPermission(INTERACT_ACROSS_USERS_FULL,
         null);
```

```
4       enforceShellRestriction(DISALLOW_DEBUGGING_FEATURES,
            callingUid, ...);
5       ...
6       if (intentFilter.countActions() == 0) {
7           Slog.m232w(TAG, "Cannot set a crossProfile intent filter
                with no filter actions");
8           return;
9       }
10      ...
11      resolver.addFilter(newFilter);
```

Listing 4.1: addCrossProfileIntentFilter

As shown, the API consists of a few execution paths, including the following[2]:

1. Path 1: a permission enforcement, a shell restriction enforcement, an input validation (conditional statement at line 6), a method invocation `Slog.m232w` method, and a return statement.

2. Path 2: a permission enforcement, a shell restriction enforcement, an input validation (conditional statement at line 6), and a method invocation `addFilter`.

We observe that Path 1 denotes a failed input validation while Path 2 implements the actual functionality provided by the API. As such, the permission enforcement and shell restrictions requirements in Path 1 *are not related* to the reachabled resources (i.e., `Slog.m232w` and the return statement).

This observation leads us to propose a reduction technique to remove such noisy paths from our training dataset. Particularly, we identify and remove paths that do not contribute to the core functionality of the system service API. Since it is not straightforward to define *contribution* due to its subjective nature, we rely on the following rules to pinpoint paths depicting common error handling logic:

Failed input validations. Different from general conditional statements, an input validation has the following two unique properties (1) it compares input with predefined constants or dynamically derived results from other methods, and (2) terminates the execution shortly after the validation fails. The termination may be carried out after a few operations for post-failure diagnosis and/or cleanup. By leveraging these observations, we

[2]The actual implementation of the API includes more paths which we have omitted to ease readability

automatically identify failed input validation paths as follows: First, for each path, we inspect all conditional statements to identify those where at least one of the operands is a parameter. Second, we ensure that the path corresponds to a failed branch by checking if it leads to some exception (i.e., `IllegalArgumentException`) or a return with a constant (i.e., error code) such that there is no other statement along the path except of cleanup operations (e.g., public interface recycling) and diagnostic operations (e.g., log statements). The failed validation path may legitimately include other statements for constructing inputs to a cleanup / diagnostic operation, so we further tolerate its transitively data-dependent instructions.

Failed access control paths. We leverage the following observations to automatically identify failed access control paths: (1) the path should include an access control check, and (2) the API terminates the execution shortly after failure. The termination action is similar in nature to that of a failed input validation. Thus, we repurposed our detection procedure of failed input validation paths to tackle this case. More details on how we identify access control checks are discussed later in Section 4.5.

We demonstrate the benefit of our path reduction technique in Section 5.3.

IR Instruction	**Transformation**
SSAAbstractInvokeInstruction	methodName (var1, ..., varM) ;
SSAGetInstruction	varType varName = object.fieldName;
SSACheckCastInstruction	varType1 var1 = (varType2) var2;
SSAPutInstruction	object.fieldName = var1;
SSABinaryOpInstruction	varType varName = var1 [binary_operator] var2;
SSAReturnInstruction	returnType var1;
SSANewInstruction	varType varName = new varType (var1, ... , varM);

Table 4.1: IR Transformation Guide (*varN* is recursively resolved)

4.4 Phase-4: IR Decompilation

At this stage, the program statements in the collected execution paths are in WALA Intermediate Representation (IR). IR abstracts away a lot of the source code constructs, which can be helpful for the model to understand structural and semantic dependencies. Furthermore, Gallagher et al. have shown that CodeBert performs better on *C* source code than its IR representation (LLVM) for the task of vulnerability detection [14]. This representation is not amenable to CodeBert, which has been pre-trained in source code and documentation [29]. Thus, it is necessary to transform the IR statements to a suitable representation before fine-tuning can be applied. To tackle this challenge, we devise a decompilation technique for WALA IR that preserves general source code constructs such as method invocations, assignments, and binary operations. Specifically, we adopt a set of transformation rules (shown in Table 4.1) to convert IR execution paths into abstracted execution paths. The latter are akin to a pseudo-source code representation, which CodeBert can understand, as they maintain key constructs and structure found in source code.

```
1. getfield < Application,
   Lcom/android/server/pm/PackageManagerService, mContext, \
                    <Application,Landroid/content/Context> > v1
2. invokevirtual < Application, Landroid/content/Context,
   enforceCallingOrSelfPermission(
                    Ljava/lang/String;Ljava/lang/String;)V >
                       v7,v8,v9:#0 @5 exception:v10
3. invokestatic < Application, Landroid/os/Binder,
   getCallingUid()I > v12:@8 exception:v11
4. invokestatic < Application,
   Lcom/android/server/pm/PackageManagerServiceUtils, \
                enforceShellRestriction(Ljava/lang/String;II)V >
                v14,v12,v4 @18 exception:v15
5. invokevirtual < Application, Landroid/content/IntentFilter,
   countActions()I > v17: v2 @21 exception:v16
6. conditional branch(ne, to iindex=-1) v17,v9
7. invokevirtual < Application, Lcom/android/server/pm/Settings, \
        editCrossProfileIntentResolverLPw(I) \
        CrossProfileIntentResolver; > v23:v21,v4 @45 exception:v22
8. new <Application,.../pm/CrossProfileIntentFilter>@38:v19
9. invokevirtual < Application, .../pm/CrossProfileIntentResolver,
                    addFilter(Landroid/content/IntentFilter;)V >
```

```
v23,v19 @79 exception:v40
```

Listing 4.2: IR Execution Path: Path-1

```
[get]: this.mContext;
[inv]: var7.enforceCallingOrSelfPermission(var8,param2);
[inv]: Binder.getCallingUid();
[inv]: PackageManagerServiceUtils. \
                    enforceShellRestriction(var14,var12,param4);
[inv]: countActions();
[Con-fail]: if (var17 != var9)
[inv]: this.mHandler.editCrossProfileIntentResolverLPw(param4);
[new]: new CrossProfileIntentFilter();
[inv]: var23.addFilter(var19);
```

Listing 4.3: Abstract Execution Path: Path-1

Consider the unmodified execution path (2) and its corresponding abstracted path depicted in the listings 4.2 and 4.3. As shown, method invocations are abstracted into the form `returnType varName = methodName (var1, ..., varM)` , return statements in the form `returnType var1`, etc. The abstracted forms are substantially shorter, with a reduced number of possible tokens; yet expressive enough to preserve the original logic.

Incorporating data flow information. Program statements may exhibit different behavior depending on supplied parameters. As such, it is crucial that the model learns such contextual information to properly infer its access control requirement. We noticed though that our model was not sufficiently able to observe this contextual information from the abstracted paths, which we speculate is due to the indirect nature of certain data flows. Therefore, we augment the abstracted paths with data flow information. In particular, we leverage interprocedural def-use chains to transitively resolve indirect data flows. We note that our analysis cannot resolve certain flows (e.g., a variable that may hold different values depending on certain conditions). We differentiate those unresolved values from parameters (which cannot be resolved inherently through static analysis) by using different place holders during augmentation. In particular, we use the place holder `PARAM_index` to denote a parameter in the abstracted paths and `UNRESOLVED` to denote an unresolved value in the rest of the cases.

```
[inv]: this.mContext.enforceCallingOrSelfPermission(
                            INTERACT_ACROSS_USERS_FULL,0);
```

```
[inv]: int var12 = Binder.getCallingUid();
[inv]: PackageManagerServiceUtils.enforceShellRestriction(
                no_debugging_features, var12, param4);
[inv]: int var17 = countActions();
[Con-fail]: if (var17 != 0)
[inv]: CrossProfileIntentResolver var23 =
   var21.editCrossProfileIntentResolverLPw(param4);
[new]: CrossProfileIntentFilter var19 = new
   CrossProfileIntentFilter();
[inv]: var23.addFilter(var19);
```

Listing 4.4: Execution Path with Data Resolution: Path-1

Our model demonstrates a significant improvement in accuracy when using our proposed IR decompilation. More details are discussed in Section 5.3.

4.5 Phase-5: Label Extraction

As mentioned earlier, access control labels are extracted at the granularity of execution paths, rather than the whole API. We traverse the statements along each identified path. When a conditional statement is encountered, we inspect it to check if it is related to access control using the patterns defined in Kratos [27] and AceDroid [3]. For example, if one operand in the predicate is evaluated to an invocation of `checkPermission`, we tag the statement as an access control. We then extract more information related to these checks using `DefUse` chains, including the operator, operand values, if any, and parameters to operands when applicable. For example, if the operand is an invoke statement for the method `checkPermission`, we resolve its concrete parameter permission.

If multiple access control-related checks are identified along the same path, our solution merges them using an AND. The resulting union of the identified checks is further normalized using the procedure proposed in AceDroid [3]. Normalization (privilege-perspective) yields four labels *Signature*, *Dangerous*, *Normal* and *No Protection*.

Merging access control classes. Application of the AceDroid normalization procedure in our collected training corpus led to significant class imbalance. In particular, the normalized labels *Normal* and *Dangerous* had substantially fewer samples compared to the other labels. To address the issue, we group the labels *Normal* and *No Protection* under the class *No Protection*; since normal permissions are granted to applications without user intervention. Similarly, we group *Dangerous* and *Signature* labels into the class *Protected*

– since *Dangerous* permissions require explicit user approval. As such, our overarching access control recommendation is translated into a binary classification problem, that is, whether an API requires access control.

Chapter 5

Model Evaluation and Experiments

Our evaluation experiments are designed to answer the following research questions:

- **RQ1:** Can the (baseline and combined) model correctly predict the access control requirement for Android APIs?
- **RQ2:** What is the impact of various design decisions on the model performance?
- **RQ3:** What is the runtime and memory overhead of the recommendation pipeline?
- **RQ4:** Can the model predict access control inconsistencies?
- **RQ5:** How effective is the model compared to results from Poirot?

5.1 Experimental Setup

We employ the static analysis components of our tool with the help of WALA [2] and leverage Akka-Typed [1] for parallelization. We construct CodeBERT using PyTorch and fine-tune it on an Intel Xeon CPU @2.20 GHz with 12GB Ram and Tesla T4 GPU, with a learning rate (lr) of 0.0001 for ten epochs and an early stopping strategy to prevent overfitting. We use the binary cross-entropy loss function and the 'Adam optimizer to modify the model's weights.

5.1.1 Training Samples Collection

Our training corpus for fine-tuning CodeBERT was obtained by analyzing three AOSP ROMs (versions 11, 12, and 13). Row 2 of Table 5.1 lists the statistics – Column #2 reflects the number of unique APIs recovered by analyzing the three ROMs. Since the ROMs share the majority of APIs, we remove duplicate ones and merge inconsistencies by prioritizing the latest version.

Column #3 shows the resulting number of data points. Observe that the recovered data points reflect the total number of enhanced execution paths in the APIs, excluding those corresponding to common error handling logic (see 4.3). Recall that we enhanced the execution path to represent the path's subsequences (see more details in Section 4.2).

ROM	**# (Unique) APIs**	**# Data Points**	**# 'Requires Protection'**	**# 'No Protection'**
AOSP (11, 12, 13)	3529	20145	14903 (73.9%)	5242 (26%)
Xiaomi Civi 1S	811	6539	4258 (65.1%)	2281 (34.8%)
Xiaomi POCO	765	3739	2196 (58.7%)	1543 (41.2%)
Lenovo TB300FU	708	4085	2264 (55.4%)	1821 (44.5%)
Amazon Fire HD	698	5809	3056 (52.6%)	2753 (47.2%)

Table 5.1: Android ROM Statistics

Our analysis of AOSP yielded 20,145 labeled data points, with 14,903 (73.9%) protected samples and 5242 (26%) unprotected samples. Class imbalance can lead to low precision, over-fitting, and loss of information about the minority class during fine-tuning. To address the imbalance, we use a combination of *task-specific* and *MixUp* data augmentation techniques [28].

Specifically, we combine *unprotected* execution paths belonging to different APIs to generate a synthetic *unprotected* training sample. This operates under the intuition that the combination of two unprotected paths would also not require protection.

5.1.2 Training samples vs. testing samples

We utilize 10-fold cross-validation during model training to evaluate the model's performance on the dataset curated from AOSP. The results of cross-validation support the generalizability of the model on unseen data. During each iteration of the cross-validation process, we divide the training dataset into ten folds. For each iteration, 9 out of the ten

folds act as training data, and the remaining fold acts as validation data. This process is repeated ten times, ensuring that the model is trained and evaluated on different subsets of the dataset.

5.1.3 Evaluation Metrics

To evaluate the performance of our model, we use four metrics 'accuracy,' 'precision,' 'f1-score', and 'recall' [19]. To obtain the best model, we monitor these metrics during the training phase.

Modification	**Accuracy**	**Precision**	**Recall**	**F1-score**
Baseline	73%	78%	76%	77%
Feature Reduction	77%	79%	75%	77%
Path Decomposition	79%	81%	76%	78%
IR Decompilation	82%	83%	79%	81%
Synthetic Data	85%	83%	88%	85%
Combined	**93%**	**91%**	**92%**	**91%**

Table 5.2: Model performance under different design decisions adopted individually and collectively

5.2 RQ1: Baseline and Combined Model Performance

Here, we report the predication results of our baseline model, where we only employ individual execution traces (without decomposition) and their labels as data points. Row 1 in Table 5.2 reports the performance results. As shown, the baseline model achieves mediocre performance (73% accuracy). This means that the model cannot sufficiently learn the correspondence between the extracted code paths and access control labels without our optimizations. However, as we will show in RQ2 results, the combined model, where we adopt all our proposed design decisions, achieves a significantly higher accuracy. Next, we discuss these improvements.

5.3 RQ2: Impact of Various Design Decisions on Model Performance

To evaluate the performance gain of each individual design decision (Section 4), we conducted five experiments using the AOSP dataset. In each experiment, we turn on an individual design decision and turn off the rest. For each iteration, we re-evaluate the (modified) model's performance using a 10-fold cross-validation. The experiments led to the results reported in rows 3-6 in Table 5.2.

We observe that each individual feature leads to an improvement in (almost) all metrics. Certain features resulted in better gains than others. Our proposed decompilaiton of the program instructions from WALA's IR to the abstracted form led to a substantial increase in the accuracy metric; from 0.73 to 0.82. Besides, adding the *Not Protected* synthetic data to reduce class imbalance led to a significant gain in terms of all four metrics (see Section 5.3)

As shown in the last row of Table 5.2, optimum performance is obtained by combining all design decisions; all metrics exhibit a notable gain.

5.4 RQ3: Runtime and Memory Overhead

Table 5.3 reports the time and memory overhead required to fine-tune CodeBert using the AOSP ROMs collected. The results are broken down in two stages:

1. Static analysis: This reflects the analysis required to collect and process the features and extract corresponding labels to fine-tuning CodeBert.
2. Training and evaluation : this reflect fine-tuning and evaluating the model using 10-fold cross validation.

As shown, static analysis takes 148 minutes and 510 mb of memory, while training and evaluation take more than 12 hours (790 minutes). Note that this is a one-time process, and hence the overhead is acceptable.

ROM (Android Version)	**# Detected Inconsistencies**	**# Manually Examined**	**# Manually confirmed (Previously Known)**
Amazon Fire HD (10)	31	10	8 (5)
Xiaomi Poco C3 (10)	25	5	3 (2)
Xiaomi Civi 1S (11)	5	3	2
Lenovo TB300FU (12)	6	2	1

Table 5.4: Inconsistencies Landscape in Vendor-Customized ROMs.

Metrics	**Static Analysis**	**Fine-Tuning**
Memory	510	5800
Time	148	790

Table 5.3: Runtime and memory overhead for fine-tuning CodeBert using AOSP ROMs (Time is in minutes, and memory is in MB)

5.5 RQ4: Detecting Access Control Inconsistencies

Recall that our overarching goal is to fine-tune CodeBert to learn access control recommendations for a given (custom) API. We can detect access control inconsistencies by comparing them against the actual implementations. Here, we demonstrate this ability on four custom ROMs from three vendors: Xiaomi Civi 1S (ver. 11) and Poco C3 (ver. 10), Lenovo TB300 (ver. 12), and Amazon Fire HD 10 (ver. 10). Rows 3-6 in Table 5.1 show the pertaining statistics.

Our target APIs for access control recommendation correspond exclusively to *non-protected* APIs; i.e., those for which static analysis identifies a lack of access control enforcement.

To aid the model in learning vendor-specific semantic properties, we augment our training data with the rest of the protected vendor APIs, since they are not used as targets. We highlight that we only consider *protected* data points from custom ROMs for training (i.e., execution paths with access control enforcement); since unlike AOSP, we cannot assume that *unprotected* data points are correctly implemented.

Results. Table 5.4 presents the inconsistency landscape as detected by the fine-tuned model. Column #1 reports the total number of inconsistencies for each custom ROM;

they range from as low as 5 in Xioami Civi 1S (ver. 12) to as high as 31 in Amazon Fire HD (ver. 10). Observe that due to the lack of ground truth, it is difficult to evaluate the validity of reports. Therefore, we resorted to manual examination (as performed by related work [27, 3]). Specifically, two authors investigated a subset of reports. Note that not all reports could be understood due to the proprietary nature of custom APIs, and therefore only a subset could be examined as shown in Column #3 in the table. Manual validation led to the confirmation of the cases listed in Column #4, ranging from 1 in Lenovo TB300FU (ver. 11) to 8 in Amazon Fire HD (ver. 10). In total, of the 20 manually examined cases, 14 were confirmed to be indeed inconsistent. Of the 14 cases, 7 have been previously reported by Poirot [10]. We reported the remaining 7 cases to the corresponding vendors; at the time of writing, 4 have been acknowledged, 2 are currently being reviewed and 1 was deemed informative (but not harmful)[1].

Estimated False Positive Rate. Previous approaches estimate False Positive (FP) Rate by manually investigating a sample of reported inconsistencies and detecting the ones which lead to exploitable vulnerabilities [10]. Our manual investigation of the reported inconsistencies suggest that 6 of the 20 cases are false positives. This implies that 30% is our (estimated) false positive rate. Due to the lack of ground truth and difficulty in understanding proprietary code, it is difficult to validate the rate at a larger scale.

Note: the aforementioned Estimated False Positive Rate represents the ability of our model to detect exploitable vulnerabilities based on reported inconsistencies. This is different from the precision and f1-score from the evaluation of the model in RQ1 5.2, which represents the ability of our model to report inconsistencies.

Access control recommendation overhead. Table 5.5 shows the time and memory overhead incurred by our model to predict access control recommendations for custom APIs. The static analysis stage reflects the analysis required to extract various features from the custom (*unprotected* APIs for which prediction is to be made. Our tool takes less than a minute to predict a recommendation for all APIs.

[1]The inconsistency allows accessing a protected file, which does not exist anymore in the updated firmware

Android ROM	# APIs	Static Analysis		Prediction	
		Time	Mem	Time	Mem
Xiaomi Civi 1S	811	24	225	<1	2200
Xiaomi POCO	765	22	232	<1	2100
Lenovo TB300FU	708	21	200	<1	2200
Amazon Fire HD	698	35	200	<1	2000

Table 5.5: Access control recommendation overhead for custom ROMs (Time is in minutes, and Memory is in MB)

5.6 RQ5: Comparison to Poirot

A closely-related work to our proposed approach is Poirot [10], which projects access control recommendations though probabilistic inference. Our work is motivated by Poirot's success in using code patterns to infer access control requirement. Here, we compare our model's accuracy against Poirot and demonstrate the unique benefits of using our approach.

Comparison of Model Accuracy with Poirot. Poirot [10] reports a maximum accuracy of 82.7% when analyzing AOSP ROMs. Our model achieves an improved accuracy rate of 93% as shown in Table 5.6.

Approach	Accuracy	Coverage
Poirot: *(Cutoff 0.80)*	75.3%	60.2%
Poirot: *(Cutoff 0.90)*	77.4%	59.4%
Poirot: *(Cutoff 0.95)*	82.7%	55.6%
CodeBert	**93%**	**100%**

Table 5.6: Evaluation of Recommendation Accuracy (Utilizing 90% of AOSP Data for Training/Rule Extraction Purposes)

Comparison of Estimated False Positive Rate with Poirot on custom ROMs. Similar to our approach, Poirot estimates false positives for custom vendor APIs through manual investigation. Out of the discovered inconsistencies, it reports 32.7% as false alarms. This is comparable to our estimated False Positive Rate of 30%.

Coverage Comparison with Poirot. As mentioned in the Motivation Section 2, Poirot's approach is inherently susceptible to the number of collected observations. Namely, a lower

number of observations can hinder its ability to make confident recommendations. As reported in the paper, this limitation has led to a relatively lower coverage, ranging from 55. 6% to 60. 2%, as shown in Table 5.6. In contrast, our approach does not suffer from this limitation; the model can predict recommendations for all (seen & unseen) APIs.

Discovered Inconsistencies. We acquired the list of reported inconsistencies by Poirot for Amazon Fire HD and Xiaomi Poco C3[2], and compared it with the inconsistencies detected by our model. We confirm that our tool can discover all of Poirot's detected inconsistencies (7 in total).

[2]We purposely analyzed these two ROMs to allow comparison with Poirot; we could not correctly decompile the other LG ROM by our tool chain

Chapter 6

Discussions

6.1 Threats to Validity

6.1.1 Reliance on AOSP

The accuracy of our dataset used to train the model could be a potential issue that could compromise its validity. An accurate mapping of code properties to access control requirement is not readily available due to the lack of security specification for Android APIs. To achieve ground truth for model training, we used the data collected from AOSP as a reference. Although this decision is in line with the body of work on Android access control evaluation, this does not guarantee that AOSP is free of inconsistencies and errors in access control. Moreover, we assume that the data collected from AOSP is reflective of the access control enforcement in the vendor-customized ROMs. To ensure that our model is effective on non-AOSP ROMs, we take a conservative approach and assume that the *Protected* APIs in vendor-customized ROMs are reliable. Therefore, we frequently update our model on unseen vendor ROMs (by combining the *Unprotected* class from AOSP and the *Protected* class from the vendor-customized ROM) and generate predictions on custom *Unprotected* APIs defined in the vendor ROM. We only freeze the update if the performance in the validation set increases.

6.1.2 We Assume that Combining Two Unprotected Paths Retains Unprotected Status

To generate synthetic data examples for our dataset, we combine path data from several unprotected resources to generate a new data point. This approach assumes that we can simulate a new unprotected API by combining resources from multiple unprotected APIs. However, we acknowledge that there is an implicit risk in such a synthesis. Although the likelihood remains low, there is still a chance that not all synthetically generated entry points retain their unprotected status in real-world scenarios. This serves to remind us that synthetic data, while highly beneficial, should be employed with an understanding of its implicit assumptions and potential restrictions.

6.2 Limitations

6.2.1 False Positives due to internal and implicit access checks

Due to the diverse nature of access control checks, detecting and modeling them as labels can cause problems for the deep learning model. Therefore, to generate labels for the training and testing data, we employ access control detection and normalization techniques to standardized the diverse access checks. However, the static analysis tools struggle to detect two prominent cases: implicit access control checks and internal access control, which can affect our dataset's reliability. Addressing the accurate detection of implicit access control checks is beyond the scope of this work.

6.2.2 Inability to evaluate our model on vendor customized ROMs

To evaluate our approach, we performed a 10-fold cross-validation on AOSP data. However, we were unable to test our model's performance on vendor-customized Android ROMs using the deep learning evaluation metrics (*accuracy, precision, recall and f1-score*). This is because we assume that AOSP is reliable and use it as training data; however, vendor-customized ROMs lack the ground truth labels, which are necessary to evaluate the performance of our model. Instead, we evaluated our model on vendor-customized ROMs by manually analyzing the flagged APIs and noting the cases of potential inconsistencies.

Chapter 7

Related Work

7.1 Deep Learning for Software Engineering

In literature, several software engineering tasks have adapted machine learning and deep learning approaches to achieve state-of-the-art performance for their respective tasks [32]. For example, Feng et al. proposed CodeBERT, a bimodal pre-trained model for natural language and source code [12]. In their work, Zhou et al. demonstrate that fine-tuned CodeBERT performs well for unseen data in code summarization, code-to-document generation tasks, and new tasks that the model was not trained on, such as defect prediction tasks. [35]. To our knowledge, deep learning approaches have yet to be extensively investigated for access control recommendation as a task. The access control check's diverse nature and the ground truth data's limited availability could be possible reasons. Nonetheless, we provide some insight into the vulnerability detection task, as there is a close relationship between vulnerability detection and access control recommendation regarding the necessary features for the decision-making process of the deep learning model. Researchers have adapted deep learning models for vulnerability detection to improve the static and dynamic analysis-based tools [24, 37, 25, 8]. VulDeePecker implements BiLSTM networks to implement code attention which allows them to discover and pinpoint the location of vulnerabilities in source code [22]. SySeVR uses *Bidirectional Gated Recurrent Unit* (BGRU) in conjunction with a systematic framework for extracting and encoding syntactic and semantic features from C/C++ source code into vector representations to discover vulnerabilities [21]. In their work, Zhou et al. propose *Devign*, which uses graph neural network to locate vulnerabilities in C projects using Abstract Syntax Trees (AST) enhanced with control and data flow information as features [36]. Kim et al. fine-tune Bert

to detect system vulnerabilities in C/C++ source code [19]. Instruction2Vec encourages us to perform IR decompilation to enhance the performance of our model. Specifically, Lee et al. [20] used binary code modelling to generate assembly instructions, taking advantage of the syntax of the code. Gallagher et al. motivate us further to implement *IR Abstraction* by showing that CodeBert performs better on C source code compared to it's IR representation [14].

7.2 Inconsistency Analysis

Kratos is a static analysis tool that discovers inconsistencies in the hard-coded access control checks along the path to sensitive resources [27]. Due to the diverse nature of access control checks, Kratos suffers from high false positive rates. AceDroid improves upon the previous convergence analysis tools by normalizing access control checks and converting diverse access control checks into their canonical form to reduce false positives [3]. ACMiner uses text analysis techniques to statically detect potential authorization checks and discover access control inconsistencies without oversimplifying access control checks as compared to AceDroid [16]. Doing so allows ACMiner to detect vulnerabilities more precisely when analyzing single system images. IAceFinder [34] analyse sensitive APIs and Intent objects to detect leakage of sensitive data. FRED [17] enhances the convergence-based inconsistency by identifying mappings between Remote Procedure Call (RPC) entry points and concrete file paths in the Android ROM to detect permission re-delegation vulnerabilities. Although all of the mentioned inconsistency detection approaches simulate access control checks in distinct ways, their underlying mechanism remains the same. Poirot [10] demonstrates that convergence analysis suffers from high false positives due to inaccurate resource access control mapping. Furthermore, they introduce Poirot, a static analysis tool that uses probabilistic inference to generate access control recommendations for system APIs based on pre-defined heuristics.

Chapter 8

Conclusion

In this paper, we introduce a deep learning-based access control recommendation approach that predicts access control requirements for Android APIs. Our approach can automatically learn and fine-tune contextual relations in Android APIs, eliminating the need to manually describe and fine-tune rules. Through extensive evaluation, we demonstrate the promising application of our fine-tuned model: The model achieves 93% accuracy, 91% precision, and 92% recall in access control recommendations. Our approach further identifies **7** previously undiscovered access control inconsistencies, of which **4** were recognized by the respective vendors in 4 Android ROMs.

As part of the future work, one might obtain a more balanced dataset by sourcing data from AOSP and vendor-customized ROMs, specifically handpicking the system service APIs with fine-grained access control protections. Such data would require manual annotation, especially since it would incorporate system service APIs from vendor-customized Android ROMs. The curated dataset would then serve as a basis to further train and evaluate models, pushing them towards recommending more detailed access control checks. Finally, we highlight the benefits of this approach and address the unique challenges we faced while adapting a deep learning approach to recommend access control enforcement in Android, underscoring the need for continued research and advances in this critical domain.

References

[1] Akka. *https://akka.io/*, 2022.

[2] Wala. *https://github.com/wala/WALA*, 2022.

[3] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS*, 2018.

[4] 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.

[5] Esraa Saleh Alomari, Riyadh Rahef Nuiaa, Zaid Abdi Alkareem Alyasseri, Husam Jasim Mohammed, Nor Samsiah Sani, Mohd Isrul Esa, and Bashaer Abbuod Musawi. Malware detection using deep learning and correlation-based feature selection. *Symmetry*, 15(1):123, 2023.

[6] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 1101–1118, 2016.

[7] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] 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*, pages 937–950, 2022.

[11] Anam Fatima, Ritesh Maurya, Malay Kishore Dutta, Radim Burget, and Jan Masek. Android malware detection using genetic algorithm based optimized feature selection and machine learning. In *2019 42nd International conference on telecommunications and signal processing (TSP)*, pages 220–223. IEEE, 2019.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[13] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 935–947, 2022.

[14] Shannon K Gallagher, William E Klieber, David Svoboda, and CARNEGIE-MELLON UNIV PITTSBURGH PA. Llvm intermediate representation for code weakness identification. 2022.

[15] M Gopinath and Sibi Chakkaravarthy Sethuraman. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review*, 47:100529, 2023.

[16] 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*, pages 25–36, 2019.

[17] 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, 2022.

[18] Justin M Johnson and Taghi M Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6(1):1–54, 2019.

[19] Soolin Kim, Jusop Choi, Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. Vuldebert: A vulnerability detection system using bert. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 69–74. IEEE, 2022.

[20] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19):4086, 2019.

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[23] Hongliang Liang, Yuxing Yang, Lu Sun, and Lin Jiang. Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[24] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[25] Huan Mei, Guanjun Lin, Da Fang, and Jun Zhang. Detecting vulnerabilities in iot software: New hybrid model and comprehensive data analysis. *Journal of Information Security and Applications*, 74:103467, 2023.

[26] Dhruv Rathi and Rajni Jindal. Droidmark: A tool for android malware detection using taint analysis and bayesian network. *arXiv preprint arXiv:1805.06620*, 2018.

[27] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[28] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Text data augmentation for deep learning. *Journal of big Data*, 8:1–34, 2021.

[29] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. Generalizability of code clone detection on codebert. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–3, 2022.

[30] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496, 2022.

[31] R Vinayakumar, Mamoun Alazab, KP Soman, Prabaharan Poornachandran, and Sitalakshmi Venkatraman. Robust intelligent malware detection using deep learning. *IEEE access*, 7:46717–46738, 2019.

[32] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

[33] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

[34] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In *The 2022 Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[35] Xin Zhou, DongGyun Han, and David Lo. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 425–436. IEEE, 2021.

[36] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[37] Noah Ziems and Shaoen Wu. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2021.