Preventing Capability Abuse Through Systematic Analysis of Exposed Interfaces

by

Yuru Shao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2019

Doctoral Committee:

Professor Z. Morley Mao, Chair
Associate Professor Kira Barton
Assistant Professor Qi Alfred Chen, University of California, Irvine
Professor Atul Prakash

Yuru Shao

yurushao@umich.edu

ORCID iD: 0000-0001-9519-3930

To my wife and my family.

ACKNOWLEDGEMENTS

The past five years of my life is one of the most rewarding experiences I have ever had. As the end of my PhD journey is getting closer, my sentiment of gratitude is growing stronger. I would not have been able to make it without the help and support of many individuals.

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Zhuoqing Morley Mao who has always been of tremendous help at every stage of my PhD study. She always encourages us to aim high, start small, and keep working hard. Her support and insightful advice have helped me overcome many challenges in my research, and her inspiration will keep me chasing my dream. I am so fortunate to be her student, who has developed independent research skills under her training and supervision.

I would like to express my deepest appreciation to my dissertation committee, Prof. Atul Prakash, Prof. Kira Barton, and Prof. Qi Alfred Chen for their valuable feedback. Their time and input are very much appreciated.

I am incredibly grateful to my collaborators. My dissertation would not have been possible without Prof. Zhiyun Qian and Jason Ott from the University of California, Riverside. Our two-year collaboration was super enjoyable and fruitful. I also wish to thank Prof. Dawn Tilbury, Dr. James Moyne, Ilya Kovalenko, Efe Balta, and Dr. Yassine Qamsane from the Mechanical Engineering department at the University of Michigan, as well as Dr. Felipe Lopez, Dr. Zheng Zhang, and Dr. Miguel Saez, who were with the ME department. I benefited a lot from our interdisciplinary collaboration on the challenging yet intriguing Software-Defined Control project.

I would like to extend my sincere thanks to Dr. Ruowen Wang, Dr. Ahmed M. Azab, Dr. Xun Chen, and Dr. Haining Chen, my mentors and friends at Samsung Research America where I spent two wonderful summers. I also thank Dr. Chenguang Shen, Luis Delgado, and Dr. Pieter Hooimeijer at Facebook. I truly enjoyed working with them on solving real-world problems with big impact.

Special thanks go to Prof. Daniel Xiapu Luo from the Hong Kong Polytechnic University and Guojun Peng from Wuhan University. They enlightened me the first glance of system security research.

I had a great pleasure of working with my fellow labmates, David Ke Hong, Shichang Xu, Yikai Lin, Xiao Zhu, Jie You, Shengtuo Hu, Yulong Cao, Xumiao Zhang, Jiachen Sun, Won Park, Jiwon Joung, and Can Carlak. I wish you all the best of luck in your future endeavors. I also thank my dear friends, Dr. Yunhan Jia, Chao Kong, Dr. Zhe Wu, Chenxiong Qian, Scott Murphy, Dr. Yuping Li, Dr. Yihua Guo, Yunchang Xiao, Xin Huang, and many others.

And finally, last but not least, I would like to thank my family: my mother Lei Deng, my father Hailong Shao, my brother Yuting Shao, and in particular, my lovely wife, Ming Sun. They have been a constant source of love and understanding throughout this journey.

Go Blue!

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Connectivity and interoperability are becoming more and more critical in today's software and cyber-physical systems. Different components of the system can better collaborate, enabling new innovation opportunities. However, to support connectivity and interoperability, systems and applications have to expose certain capabilities, which inevitably expands their attack surfaces and increases the risk of being abused. Due to the complexity of software systems and the heterogeneity of cyber-physical systems, it is challenging to secure their exposed interfaces and completely prevent abuses. To address the problems in a proactive manner, in this dissertation, we demonstrate that systematic studies of exposed interfaces and their usage in the real world, leveraging techniques such as program analysis, can reveal design-level, implementation-level, as well as configuration-level security issues, which can help with the development of defense solutions that effectively prevent capability abuse.

This dissertation solves four problems in this space. First, we detect inconsistent security policy enforcement, a common implementation flaw. Focusing on the Android framework, we design and build a tool that compares permissions enforced on different code paths and identifies the paths enforcing weaker permissions. Second, we propose the Application Lifecycle Graph (ALG), a novel modeling approach to describing system-wide app lifecycle, to assist the detection of diehard behaviors that abuse lifecycle interfaces. We develop a lightweight runtime framework that utilizes ALG to realize fine-grained app lifecycle control. Third, we study real-world programmable logic controller programs for identifying insecure configurations that can be abused by adversaries to cause safety viola-

tions. Lastly, we conduct the first systematic security study on the usage of Unix domain sockets on Android, which reveals both implementation flaws and configuration weaknesses.

CHAPTER I

Introduction

1.1 Overview

Thanks to the advent of wireless networks, new sensing capabilities, and cheaper, more powerful computing technologies, pervasive connectivity and interoperability have drastically changed the world. We have witnessed the rise of smartphones, smart watches, Internet-of-Things (IoT), and more recently, smart home. They have quickly become a part of our life. Moreover, new technologies such as autonomous vehicles and smart manufacturing are changing the industry.

Driven by technology advancement, today's software and cyber-physical systems are becoming increasingly open and interconnected. For example, the emerging IoT devices connect the cyberspace and the physical space and thus empower rich interactions between them. Mobile devices are equipped with various sensors that bring intelligence and awareness to applications. To realize such systems, connectivity and interoperability are the essential requirements. First, to build a vital ecosystem, systems have to support the development of applications so that they can satisfy diverse user demands. Therefore, connectivity is greatly desired. Second, systems need to be extensible and customizable in order to quickly adapt to different usage scenarios, which requires a modular design and great interoperability between various components.

Interfacing is the key to enabling connectivity and interoperability. Systems, services,

and applications expose their capabilities and provide interfaces to enable rich interactions. These interfaces have various forms, including inter-process communication (IPC) endpoints, application programming interfaces (APIs), and so on. Interfaces and the capabilities they expose need to be protected, but many potential problems can open doors for capability abuse [61, 92, 80], such as the lack of strong authentication, flaws in authorization, and sketchy, permissive policies and configurations.

It is challenging to prevent capability abuse. In general, system designers cannot anticipate all abuse techniques, and loopholes could exist in design; implementation flaws are common due to developers' lack of security expertise; people often make incorrect or insecure configurations. The complexity of modern systems and the complicated interactions they support make things worse, not to mention the cumbersome process of developing huge codebases which involves efforts from multiple parties. For instance, an Android device runs code from five different sources: the Android Open Souce Project (AOSP), device vendors such as Samsung and Huawei, hardware manufacturers (*e.g.,* Qualcomm, Broadcom), cellular service providers, and application developers. If any of them fails to protect their exposed interfaces or to consider all anomaly use cases, capability abuse is made possible.

In view of these challenges, this dissertation is dedicated to preventing capability abuse through the studies of exposed interfaces. We demonstrate that *with a security mindset, we conduct a systematic analysis of exposed interfaces using program analysis techniques and runtime monitoring methods, which can (1) reveal design-level, implementation-level, and configuration-level security problems, and (2) shed light on system design improvement for preventing capability abuse.* As summarized in Table 1.1, we address four problems in this research:

1. We investigate Android system service APIs and detect inconsistent security policy enforcement that reveals severe implementation flaws. We perform differential analysis on code paths reaching the same privilege operations. Compared to prior

Table 1.1: Summary of dissertation work

Problem scope	**Exposed interfaces**	**Project**
Analysis that reveals implementation flaws	System service APIs	Discovering inconsistent security policy enforcement in the Android framework
New design that enables fine-grained control of application lifecycle	System service APIs, lifecycle entry points	A lightweight framework for fine-grained lifecycle control of Android applications
Study of insecure configurations	Data access interfaces	Towards secure configurations for real-world industrial controller programs
Both implementation and configuration issues	IPC endpoints	The misuse of Android Unix domain sockets and security implications

work, our approach requires no policy input. We overcome challenges in obtaining a complete list of services and building a precise framework call graph, by automating entry point generation and creating IPC shortcuts.

2. Focusing on abuses of system service APIs and lifecycle entry points, we design a lightweight framework that provides fine-grained application lifecycle control. To realize it without incurring perceptible overhead, we tackle challenges in precisely identifying caller components, implementing efficient, asynchronized lifecycle hooks that do block normal application execution, and realizing non-disruptive shutdown of application components.

3. With an emphasis on data access interfaces of smart industrial control systems, we identify insecure configurations in controller programs. The challenges we manage to address include handling non-standard, vendor-specific program instructions, modeling controller scan cycle, and gathering safety-critical hardware output.

4. Targeting Android's native IPC endpoints, we find both implementation and configuration issues in using Unix domain sockets among system daemons and applications. We develop a tool that automates the discovery of vulnerable Unix domain socket usage. We overcome challenges in recognizing socket addresses, detecting authentication checks, and performing data flow analysis on native code.

1.2 Contributions

To the best of our knowledge, we are the first to systematically study exposed interfaces across software and cyber-physical systems from design, implementation, and configuration perspectives. We develop techniques to (1) automate the detection of insufficiently protected interfaces, (2) discover vulnerabilities, and (3) proactively prevent abuses. We also propose practical design, implementation, and configuration improvements to better protect exposed interfaces and effectively prevent capability abuse. In total, we have studied more than 25,000 Android applications, over 400 real-world PLC programs, and six different Android framework codebases. We have discovered 11 zero-day AOSP vulnerabilities, eight vendor customization vulnerabilities, and more than 40 vulnerable applications. Our research has reached out to AOSP, device vendors, as well as the application developer community. They all confirmed our research findings, acknowledged our efforts, and provided positive responses in fixing vulnerabilities and improving the security of their products.

1.3 Outline

The dissertation is organized as follows. Chapter II provides sufficient background and related work of the problems we attempt to solve. Chapter III presents our study on inconsistent security policy enforcement in the Android framework. Chapter IV describes a lightweight framework for fine-grained control of application lifecycle. Chapter V presents our study on access configurations of real-world programmable controller programs. Chapter VI discusses the misuse of Android Unix domain sockets and security implications. We conclude our work in Chapter VII, where we also discuss the lessons we learned and future work directions.

CHAPTER II

Background and Related Work

2.1 Capability and Capability Abuse

In our definition, a capability is the ability to perform a specific operation, for example, taking a picture, making a phone call, and modifying a system parameter. Capability abuse is the use of a capability in an unintended way for achieving harmful or malicious goals. Note that our definition is different from the concept of capability [79] that was initially introduced by Dennis and Van Horn as "A capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system."

In systems and applications, capabilities are exposed to other parties' use in the form of interfaces, including APIs, IPC endpoints, and execution entry points. For instance, the Android framework provides camera APIs that allow applications to take pictures and record videos. Access control is widely adopted by modern systems, which can protect interfaces so that processes without required privilege cannot use the capabilities exposed by those interfaces. Security policies are used to define processes' privileges. However, policy configurations may have weaknesses or even errors, resulting in security violations.

Security policy violation detection and verification. Quite a few of program analysis and verification tools have been developed to verify security properties or to detect security vulnerabilities caused by policy violation [59, 72, 82, 88, 133]. All of them, except for

AutoISES [133], require developers or users to provide code-level security policies. AutoISES can automatically infer security specifications with the input of a security check function list, and leverage inferred specifications to detect security violations automatically. However, the inaccuracy of inferred specifications can lead to false positives, and an incomplete list of security check functions will cause false negatives. Depending on different implementations, identifying security check functions could be non-trivial.

Automated security policy enforcement. To date, a few automated solutions have been proposed to address flawed security policy enforcement. Ganapathy *et al.* [87] presented a technique for automatic placement of authorization hooks, and applied it to the Linux security modules framework. Muthukumaran *et al.* [116] proposed an automated hook placement approach that is motivated by an observation that the deliberate choices made by clients for objects from server collections and for processing those objects must all be authorized. However, these approaches are highly dependent on the platforms, meaning that they cannot be easily adopted by other systems with a different security model.

2.2 Android System Services

System services implement the fundamental features within Android, including the display and touch screen support, telephony, and network connectivity. The number of services has slowly increased with each version: growing from 73 in Android 4.4 to 94 for M Preview. Most system services are implemented in Java with certain foundational services written in native code. At runtime, system services are running in several system processes, such as `system_server`, `mediaserver`. To provide functionality to other services and apps, each system service exposes a set of interfaces accessible from other services and apps through remote procedure calls. For simplicity, we define users (either another service or an app) of a system service as its *clients*. From a client's perspective, calling remote interfaces of a system service is equivalent to calling its local methods.

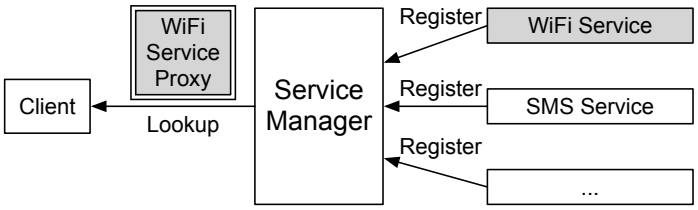

Figure 2.1: System services register themselves to Service Manager and clients call their remote interfaces with proxies.

Figure 2.1 depicts how system services are managed and used. When the Android runtime boots, system server registers system services to the Service Manager, which runs in an independent process `servicemanager` and governs all system services. When a client wants to call a system service, *e.g., Wi-Fi Service* in Figure 2.1, it first queries the service index provided by Service Manager. If the service exists, Service Manager returns a proxy object, *Wi-Fi Service Proxy*, through which the client invokes Wi-Fi Service methods. The "contract" that both the Wi-Fi Service and the proxy agree upon is defined by Android Interface Definition Language (AIDL). Figure 2.2 uses the Wi-Fi Service as an example and shows how the service and its proxy use the AIDL to define consistent interfaces.

During the compiling process, a class `IWifiManager` is automatically generated from the corresponding AIDL file `IWifiManager.aidl`. It has two inner classes, `IWifiManager$Stub$Proxy` and `IWifiManager$Stub`. All interfaces defined in the AIDL file `IWifiManager.aidl` are declared in `IWifiManager`, `IWifiManager$Stub` and `IWifiManager$Stub$Proxy`. The service extending `IWifiManager$Stub` is responsible for implementing methods defined in `IWifiManager.aidl`. Clients who wish to access service functionality only need to obtain a reference to `IWifiManager` and invoke `IWifiManager`'s method. As a result, the corresponding implementation in the service will be called. The intermediate procedure is handled by Binder IPC [5] and completely transparent to clients and services.

```
// Client.java
public class Client {
  IWifiManager mgr = IWifiManager.Stub.asInterface(
    ServiceManager.getService("wifi"));
  mgr.removeNetwork(netId);
  mgr.disableNetwork(netId);
  ...
}
// WifiManager.aidl
interface IWifiManager {
  boolean removeNetwork(int netId);
  boolean disableNetwork(int netId);
  void connect();
  void disconnect();
  ...
}
// Decompiled from IWifiManager.class
public interface IWifiManager extends IInterface {
  boolean removeNetwork(int netId);
  boolean disableNetwork(int netId);
  void connect();
  void disconnect();
  ...
}
// WifiService.java
public class WifiService extends IWifiManager.Stub {
  @override
  public boolean removeNetwork(int netId) {
    enforceChangePermission();
    ...
  }
  @override
  public boolean disableNetwork(int netId) {
    enforceChangePermission();
    ...
  }
  ...
}
```

Figure 2.2: Defining service interfaces using AIDL

2.3 Android Security and Application Analysis

The Android platform consists of multiple layers. One of Android's design goals is to provide a secure platform so that "[S]ecurity-savvy developers can easily work with and rely on flexible security controls. Developers less familiar with security practices will be protected by safe defaults." [8] All applications on Android run in an application sandbox. By default, an application can only access a very limited range of resources. The use of capabilities is regulated through a permission-based access control mechanism. Specifically, applications define the capabilities they need in their manifest files, and they have to be granted required permissions before they can access sensitive APIs. They commu-

nicate with peer apps through secure, Android-specific IPCs (*e.g.,* Binder, Intents). These Android IPC mechanisms, as documented by Google, are the preferred IPC mechanisms as they "allow you to verify the identity of the application connecting to your IPC and set security policy for each IPC mechanism." [9]

As the community continues to explore and understand Android and its ecosystem, new attacks and innovative ways of discovering vulnerabilities are being developed. Many of the existing works in Android security leverage static analysis and dynamic analysis techniques to study the framework and applications.

Static analysis tools on Android. There has been significant work in using static analysis techniques combined with call graphs to map the Android framework, understand the permission specification, understand how data is disseminated within the Android framework, and enable the functionality of gleaning information from avenues that are unassuming. PScout [57] uses static analysis tools to enumerate all permission checks within the Android framework. They are able to map all permission usages to their appropriate methods and understand the utility of permission usage within the framework. While PScout identifies permissions, they don't look at paths through the framework which would allow the invocation of the permissions they discover. Static taint analysis tools such as FlowDroid [56], AndroidLeaks [90], DroidSafe [91], and Amandroid [141] work to understand how, why, where, and what data travels through the Android framework as a user uses an application in order to perform privacy leakage detection.

Android IPC and framework vulnerabilities. Android-specific IPC mechanisms, such as Binder, Messenger, and Intents, have been thoroughly studied [76, 110, 83, 56, 90, 138]. These works aim to exploit the IPC channels in order to disclose sensitive information, *i.e.,* SMS messages, call history, and GPS data. In particular, Chin *et al.* [76] examined inter-application interactions and identified security risks in application components. They presented ComDroid to detect application communication vulnerabilities. There are also works that focus on detecting implementation flaws of the Android framework. Aafer

et al. [54] studied the threat of hanging attribute references. Unfortunately, none of the aforementioned works have explored traditional Linux IPCs on Android, *e.g.,* Unix domain sockets, as exploitable interfaces.

Security risks in customizations. Customizations to the Android framework has been known to introduce new vulnerabilities not present in the AOSP [143]. Wu *et al.* discovered that over 85% of pre-installed applications in stock images have more privileges than they need. Among them, the majority are directly from vendor customizations. They also found that many pre-installed applications and firmware are susceptible to a litany of vulnerabilities. ADDICTED [148] is a tool for automatically detecting flaws exposed by customized driver implementations. It performs dynamic analysis to correlate the operations on a security-sensitive device and its related Linux files.

Background application activities. Chen *et al.* present a study on low-level background activities of applications by looking into CPU idle and busy time [75]. They conducted a large-scale measurement study performing in-depth analysis of background application activities, quantified the amount of battery drain, and developed a metric called background to foreground correlation to measure the usefulness of background activities. [74] helps understand where and how energy drain happens in smartphones. The authors developed a hybrid utilization-based and finite state machine based model that accurately estimates energy breakdown among activities and phone components. Pathak *et al.* perform a characterization study of no-sleep energy bugs in smartphone applications and proposes a compile-time solution to automatically detect no-sleep bugs [124]. To mitigate no-sleep bugs, [137] implements a tool that verifies the absence of this kind of energy bugs with regard to a set of WakeLock specific policies using a precise, inter-procedural data flow analysis framework to enforce them. Tamer [112] is an OS mechanism that interposes on events and signals that cause task wake-ups, and allows for their detailed monitoring, filtering, and rate-limiting. It helps reduce battery drain in scenarios involving popular Android applications with background tasks.

2.4 Security of Industrial Control Systems

Industrial Control Systems (ICS) are built to electronically manage tasks efficiently and reliably. There are several types of ICS, and the most common one is the Supervisory Control and Data Acquisition (SCADA) system. SCADA systems are composed of Programmable Logic Controllers (PLCs) and other commercial hardware modules that are distributed in various locations. SCADA systems can acquire and transmit data, and are integrated with a Human-Machine Interface (HMI) that provides centralized monitoring and control for numerous process inputs and outputs. State-of-the-art ICS have adopted Ethernet-based industrial protocols, such as EtherNet/IP, to support real-time control and communications between hardware components. Moreover, industrial Internet-of-Things (IIoT) devices have become a key enabler for emerging technologies such as cloud manufacturing, industrial 4.0, and so on.

Safety analysis and verification of PLC code. Many prior efforts [93, 98, 121, 123, 67, 55, 62, 117, 119, 127] have been made to statically verify controller logic code using model checkers such as as UPPAAL [44] and NuSMV [26]. Further research efforts have also been made to conduct runtime verification in an online [89, 100] or offline manner [81, 122]. More recently, research [114, 95] has been done to enable symbolic execution on PLC code.While TSV [114] conducted static symbolic execution on its temporal execution graphs, SymPLC [95] leveraged OpenPLC [27] framework and Cloud9 engine [16] to enable dynamic analysis on PLC code.

CHAPTER III

Discovering Inconsistent Security Policy Enforcement in the Android Framework

In this chapter, we investigate Android system services APIs that expose capabilities to applications and detect inconsistent security policy enforcement, an implementation flaw that can be exploited to launch privilege escalation attacks. Using the automated detection tool we develop, system developers are able to identify vulnerabilities at an early stage, proactively preventing capability abuse.

3.1 Introduction

Access control is a well-known approach to prevent activities that could lead to a security breach. It is widely used in all modern operating systems. Linux inherits the core UNIX security model — a form of Discretionary Access Control (DAC). To provide stronger security assurance, researchers developed Security-Enhanced Linux (SELinux) [132], which incorporates Mandatory Access Control (MAC) into the Linux kernel. The fundamental question that access control seeks to answer is philosophical in nature: "who" has "what kind of access" to "what resources." It is from this single question that access control policies or security policies are derived.

Android OS employs a permission-based security model, which is a derivative of the

Access Control List (ACL) based access control mechanism [60]. In this model, an application or a user may request access to a set of resources that are governed by a set of permissions, exposed by the system or other applications.

Access control systems are known to be vulnerable to anomalies in security policies, such as inconsistency [131]. However, security policies can be inconsistent not only in their definitions but also in the ways they are enforced [133]. A significant challenge of supporting permission-based security models, as well as other access control systems, is to ensure that *all* sensitive operations on *all* objects are correctly protected by proper security checks in a consistent manner. If the proper security check is missing before a sensitive operation, an attacker with insufficient privilege may then perform the security-sensitive operation, violating user privacy or causing damages to the system. On Linux, multiple such examples have been discovered, which lead to unauthorized user account access [46], permanent data loss [133], *etc.* More recently, on the Android platform, attacks caused by inconsistent policy enforcement have also been found, *e.g.,* stealthily taking pictures in the background [73] and stealing user passwords by recording keystrokes without the necessary permissions [148]. Therefore, to ensure a safe and secure platform for users and developers, it is critical to develop a systematic approach to identify inconsistencies in security policy enforcement.

To address this problem on MAC-based operating systems, Tan *et al.* [133] present a tool, AutoISES, that can automatically infer security policies by statically analyzing source code and then using those policies to detect security violations. Its effectiveness has been demonstrated by experiments with the Linux kernel and the Xen hypervisor; however, AutoISES has several limitations that prevent it from being applied to the Android framework. First, it does not take into account inter-process communication (IPC) between different processes or threads. In Android, remote method calls across process or thread boundaries are very common. Any static analysis that fails to consider this special feature would be hugely incomplete. Second, the Android framework consists of conflated layers: Java and

C/C++, which is not currently supported by AutoISES. A fundamental limitation is that AutoISES assumes that a complete list of security check functions are given. While in Android, different types of security checks exist, making it extremely difficult to obtain a comprehensive list of them.

In view of these challenges, we propose *Kratos*, a static analysis tool for systematically detecting inconsistent security enforcement in the Android framework. Kratos accepts Java class files and security enforcement checks as input, and outputs a ranked list of inconsistencies. It first builds a precise call graph for the codebase analyzed. This call graph comprises of all the execution paths available to access sensitive resources. Each node of the call graph is then annotated with security enforcement methods that are applied to that node. For a set of entry points, Kratos compares their sub-call graphs pairwise to identify possible paths which can reach the same sensitive methods but enforce different security checks (*e.g.,* one with checks and the other without). Kratos can be applied to both the AOSP framework and vendor-specific frameworks.

Another thread of related work focuses on automated authorization hook placement to mediate all security-sensitive operations on shared resources. In work by Muthukumaran *et al.* [116], there is an implicit assumption made by the authors: they have perfect knowledge of which functions or pieces of code needs protection. They are able to automatically place the policy enforcement based on metrics such as the minimum number of checks needed. Not only do we lack this understanding, but also any understanding that might be inferred from the Android source code is further obfuscated by uncertainties introduced by developers. As illustrated in Figure 3.1, developers sometimes forget to apply security enforcement and they do not necessarily know what policies should be applied.

Kratos makes no assumptions about or attempts to infer what resources or operations in the Android framework are security-sensitive and should be protected. Instead, Kratos only identifies *where* existing security policy enforcement occurs based on observed security checks, and accurately infers security-sensitive operations by identifying inconsistency in

```
/**
 * Used by device administration to set the maximum screen off timeout.
 *
 * This method must only be called by the device administration policy manager.
 */
@Override // Binder call
public void setMaximumScreenOffTimeoutFromDeviceAdmin(int timeMs) {
    final long ident = Binder.clearCallingIdentity();
    try {
        setMaximumScreenOffTimeoutFromDeviceAdminInternal(timeMs);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

They know the use of this method should be restricted but did not apply any security checks

```
@Override
public boolean havePassword(int userId) throws RemoteException {
    // Do we need a permissions check here?
    return new File(getLockPasswordFilename(userId)).length() > 0;
}

@Override
public boolean havePattern(int userId) throws RemoteException {
    // Do we need a permissions check here?
    return new File(getLockPatternFilename(userId)).length() > 0;
}
```

Android framework developers lack knowledge of security policies that should be enforced

Figure 3.1: Code snippets from `PowerManagerService.java`.

the policy enforcement across different execution paths.

To implement Kratos, we overcome several engineering challenges. First, to maximize the completeness of our analysis, we need to cover as many system services as possible. However, system services are scattered throughout the Android framework with some implemented as private classes nested inside outer classes, making it difficult to include all service interfaces as analysis entry points. We address this by generating code that calls service interfaces and handles nested private services. Second, our analysis relies on a precise Android framework call graph, which is non-trivial to build. We tackle this challenge by resolving virtual method calls using Spark [97], a tool for performing Java points-to analysis, and applying IPC shortcuts. Third, due to the large size of the code in the Android framework, it is non-trivial to make the analysis efficient and scalable. We achieve high efficiency and scalability by optimizing the implementation and adopting a set of heuristics.

We run Kratos on six different Android codebases, including 4 versions of Android

frameworks, 4.4, 5.0, 5.1, and M preview, and two customized Android versions on AT&T HTC One and T-Mobile Samsung Galaxy Note 3 devices. After verifying the tool output manually, we find that all six codebases fail to ensure consistent policy enforcement with at least 16 to 50 inconsistencies discovered. For one, the number of inconsistencies are as high as 102. From these inconsistencies, we are able to uncover 14 highly-exploitable vulnerabilities spanning a wide range of distinct Android services on the 6 codebases; 12 of them are found to affect at least 3 codebases at the same time; 6 vulnerabilities have been patched in the latest or earlier releases but never been revealed to the public previously. All these exploits can be carried out with no permission or only low-privileged permissions (*e.g.,* the `INTERNET` permission), and lead to serious security and privacy breaches such as crashing the entire Android runtime, terminating mDNS daemon to make file sharing and multi-player gaming unusable, and setting up a HTTP proxy to intercept all web traffic on the device.

We have reported all of these vulnerabilities to the Android Security team. Among the 11 that we have received feedback, all were confirmed as low severity vulnerabilities. This indicates the challenging nature of enforcing consistent policies in a complex system like Android, and the necessity of a systematic detection tool like Kratos. Due to the lack of an up-to-date Android malware dataset, we do not have statistics on how many of these vulnerabilities have already been exploited by malicious applications in the wild.

The key contributions of this work are summarized as follows.

- Our work is the first to systematically uncover security enforcement inconsistencies in the Android framework itself, compared to previous work focusing on permission use in Android applications. We design and implement Kratos, a static analysis tool that can effectively identify inconsistent security enforcement in both the AOSP and customized Android. We tackle several engineering challenges, including automated entry point generation, IPC edges connection, and parallelization to ensure accuracy and efficiency.

- We evaluate our tool on four versions of the Android framework: 4.4, 5.0, 5.1, and M preview, as well as two customized: AT&T HTC One and T-Mobile Samsung Galaxy Note 3, and find that all codebases fail to ensure consistent policy enforcement with up to 102 inconsistencies in a single codebase. Among the discovered inconsistencies, we are able to uncover 14 highly-exploitable vulnerabilities, which can lead to serious security and privacy breaches such as crashing the entire Android runtime, ending phone calls, and setting up an HTTP proxy with no permissions or only low-privileged permissions.

- Our analysis covers all application-accessible interfaces exposed by system services implemented in Java code. Many system service interfaces are by default invisible and undocumented for applications. Among the 14 vulnerabilities we identified, 11 of them are hidden interfaces that are rather difficult to detect. These findings suggest useful ways to proactively prevent such security enforcement inconsistency include reducing service interfaces and restricting the use of Java reflection (for accessing hidden interfaces).

3.2 Motivation

In this section, we present a motivating case that demonstrates inconsistent security enforcement in Android's Wi-Fi service. We also cover various types of security enforcement adopted by the Android framework.

3.2.1 Inconsistent Security Enforcement

For convenience, we use the terms *security policy enforcement* and *security enforcement* interchangeably in this paper. Security enforcement consists of a set of security checks. *Security check* refers to a specific action which verifies whether the caller satisfies particular security requirements, *e.g.,* holds a permission or has a specific UID. The Android

framework employs several types of security enforcement: permission check, UID check, package name check, and thread status check, all explained below.

Permission checking is the most fundamental and widely used security enforcement in Android. Each app requests a set of permissions during installation. The user must allow all permissions requested or choose not to install the app. When an app calls a method exposed by a system service, the service verifies that the app holds the required permission(s). If so, the app passes the permission check and continues executing. Otherwise, the service immediately throws a security exception; as a result, the app cannot access resources guarded by the service. As shown in Figure 2.2 lines 31–34, `removeNetwork(int)` invokes `enforceChangePermission()` to check that the calling app has the `CHANGE_WIFI_STATE` permission. Permission checks are performed immediately after the code enters the service side. This is different from other systems such as SELinux, which places security checks right before accessing sensitive objects [109]. We believe these different approaches present tradeoffs in balancing the performance overhead and access control granularity.

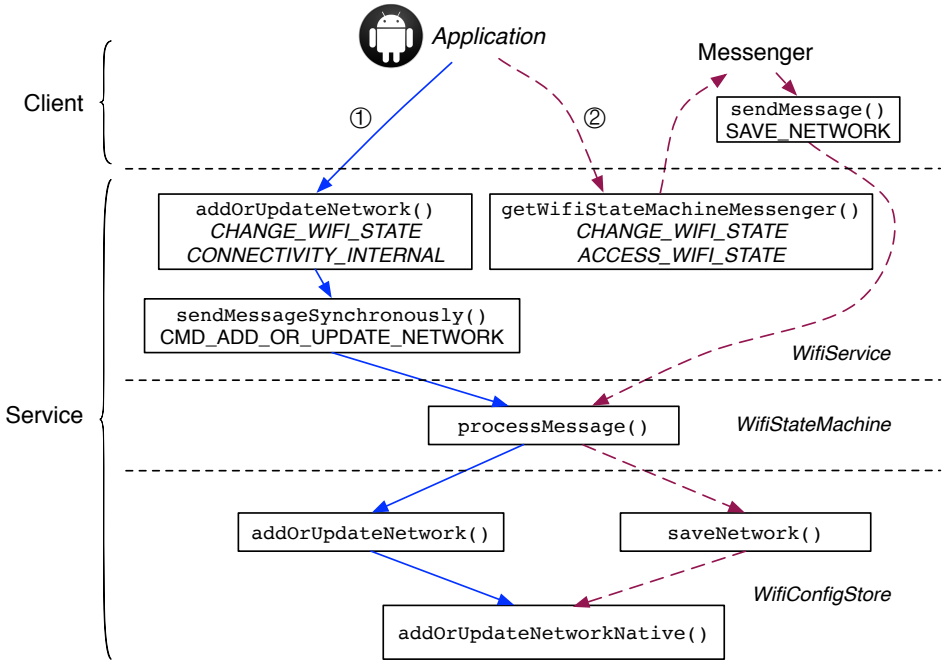

Figure 3.2: A motivating example of inconsistent security enforcement.

Despite the permission check placement, we have observed inconsistent enforcement of permissions within the same service and between services with similar functionality. For example, in Figure 3.2 we detail inconsistencies within a single service, *i.e.,* the Wi-Fi Service. It exposes two interfaces to clients: `addOrUpdateNetwork()` and `getWifiStateMachineMessenger()`. Both of them can be leveraged by apps to update Wi-Fi configurations. Though they ultimately invoke the same underlying method `WifiConfigStore.addOrUpdateNetworkNative()`, the two paths they traverse are different. The method `addOrUpdateNetwork()` first calls `sendMessageSynchronously()`, through which it sends a `CMD_ADD_OR_UPDATE_NETWORK` message to the internal Wi-Fi state machine which is able to update Wi-Fi configurations according to current status. Meanwhile, apps can call `getWifiStateMachineMessenger()` to obtain the Wi-Fi state machine's `Messenger` object, with which they are able to directly send a `SAVE_NETWORK` message to the Wi-Fi state machine to update configurations. Surprisingly, permission checks differ along these two paths. `addOrUpdateNetwork()` checks two different permissions, *i.e.,* `ACCESS_WIFI_STATE` and `CONNECTIVITY_INTERNAL`. However, only two of them are enforced in `getWifiStateMachineMessenger()`, *i.e.,* `ACCESS_WIFI_STATE` and `CHANGE_WIFI_STATE`. Considering that `CONNECTIVITY_INTERNAL` is a system-level permission, it is impossible for third-party apps to acquire it. Thus, `addOrUpdateNetwork()` is protected from third-party app usage. Nevertheless, this enforcement can be completely bypassed if an app developer, or malware author, uses `getWifiStateMachineMessenger()` instead of `addOrUpdateNetwork()`. Similarly, Telephony Service and Telecom Service both provide methods to access telephony-related functionality. While these two services are different, they do have some overlapping functionality — they expose different methods which provide similar underlying functionality, see §3.5 for further discussion.

In our analysis, we focus solely on system services. More specifically, we consider

all remote interfaces exposed by system services as application-accessible interfaces. Note that some proxy interfaces are *invisible* to apps, either because the classes are excluded from the Android SDK or because the methods are labeled with the `@hide` or `@SystemApi` javadoc directive in the source code. However, they still exist in the runtime and apps can access them using Java reflection techniques.

3.2.2 UID Check

Interfaces provided by a system service can be called by apps, as well as other system services. For some sensitive operations, the system service only allows internal uses by checking the caller's UID. As aforementioned, service interfaces are invoked through the Binder IPC mechanism. For each AIDL method call, the system keeps track of the original caller's identity in order to check it within the service side. Figure 3.3 shows a code snippet extracted from the Keyguard Service. Its `checkPermission()` method has two steps: First, it gets UID of the caller using `Binder.getCallingUid()` and verifies that the UID is equal to `SYSTEM_UID`. If so, the caller is the system and no further check is required. Otherwise, the permission check is performed.

```
1 void checkPermission() {
2   if (Binder.getCallingUid() == Process.SYSTEM_UID)
3       return;
4   // Otherwise, explicitly check for caller permission
5   if (checkCallingOrSelfPermission(PERMISSION)
6           != PERMISSION_GRANTED) {
7     ...
8   }
9 }
```

Figure 3.3: Permission check is performed if the UID check fails.

3.2.3 Package Name Check

The package name check is another means to restrict the capability of apps. For instance, in order to ensure that the client can only delete widgets that belong to itself, the

App Widget Service checks whether the caller owns the given package, by using package name check shown in Figure 3.4.

```
1 public void enforceCallFromPackage(String packageName) {
2   mAppOpsManager.checkPackage(
3         Binder.getCallingUid(), packageName);
4 }
```

Figure 3.4: Check to verify the caller owns a given package.

3.2.4 Thread Status Check

Many malicious apps are reported to run in the background and stealthily jeopardize the security and privacy of end users. This is seen through the myriad of research: stealing sensitive photos [73], inferring keystrokes [144], discovering web browsing habits [99], understanding speech through the phones gyroscope [115], *etc.* To mitigate this, Android employs thread status checks, which are designed to ensure that certain sensitive operations cannot be performed by callers running in the background. In this check, the system verifies that the caller is running in the foreground and visible to users, and considers only the operations from the foreground as performed by the user. One example of such checks in Android is implemented in the Bluetooth Manager Service. It ensures that only clients running in the foreground are able to manipulate the Bluetooth device, by checking their running status using a dedicated method named `checkIfCallerIsForegroundUser()`.

3.3 Methodology

In this section, we present our design of Kratos. We first give an overview of the design, followed by the key components that achieve the design goals.

3.3.1 Overview

Kratos' analysis flow consists of four phases, as shown in Figure 3.5. In the first phase, we retrieve Java class files of the given Android framework, and process them to generate entry points of services for further analysis. Kratos is able to analyze both AOSP and customized Android versions. Therefore, Kratos takes Java classes as input as opposed to Java source code, since customized Android frameworks are usually closed source. In the next phase, Kratos constructs a precise call graph from the entry points generated from the previous phase. Third, we annotate the framework call graph by considering different types of security checks of interest, *e.g.,* permission check, UID check, package name check, and thread status check. Each node of the call graph is examined to determine which, if any, security checks exist within it. Finally, Kratos detects inconsistencies and outputs a prioritized list of security enforcement inconsistencies.

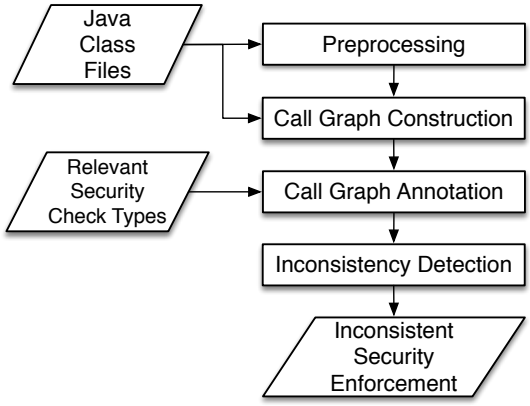

Figure 3.5: Kratos workflow.

As we have illustrated in Figure 3.2, system service interfaces with overlap in functionality may eventually invoke the same lower-level, privileged method(s) to complete their work. For instance, both of `addOrUpdateNetwork()` and `getWifiStateMachineMessenger()` exposed by the Wi-Fi Service can be used to update the configuration of the currently connected Wi-Fi network. Both methods call the `addOrUpdateNetworkNative()` method from `WifiConfigStore`. Such behav-

iors are expected — while it is common that similar high-level functionality are provided for convenience, it is not necessary for the Android framework to implement their underlying functionality multiple times, hence the convergence at the lower-level method. Based on this observation, using a call graph to represent the execution path of a service interface, we can identify where those sensitive, or lower-level, methods are invoked by any two services. This is what we refer to as an "overlap." As a result, Kratos reduces the problem of detecting security enforcement inconsistency among system service interfaces into call graph comparisons.

3.3.2 Preprocessing

The Preprocessing step collects important information for further phases. Our analysis emphasizes system services whose implementations are scattered throughout the Android framework codebase. We must first obtain a comprehensive list of *app-accessible system services* and their corresponding Java classes, from which we can retrieve all interfaces they expose that could be invoked by apps.

As we mentioned in §3.2, Service Manager manages all system services. Clients are required to obtain a proxy of the system service from Service Manager in order to invoke that service's interfaces remotely. System services that are visible for apps should be registered to *Service Manager*. In practice, besides a global service manager running in a dedicated process (`/system/bin/servicemanager`), there exist a few local service managers. System services registered to local service managers are only accessible for other services running within the same process. Therefore, we only care about system services registered to the global Service Manager since only these are accessible by apps. By looking into a service's implementation and its corresponding AIDL definition, we easily distinguish which public methods of the service are publicly accessible AIDL methods. Although apps can invoke system services directly via low-level Binder IPC mechanism without passing through the AIDL interfaces, those Binder IPC endpoints that can be reached directly are

exactly the same as those exported via the AIDL methods.

In addition to AIDL methods, we observe another type of interface exposed by system services that could be called by apps, *i.e., unprotected broadcast receivers* that are dynamically registered. Typically, system services dynamically register broadcast receivers to receive asynchronous messages from within the system. To defend against broadcast spoofing [76], either receivers should be protected by proper permissions or broadcast actions need to be protected. However, some broadcast receivers of system services are not protected at all. That means apps can also send crafted broadcasts to trigger certain method calls. Therefore, we also consider unprotected broadcast receivers in system services as app-accessible service interfaces.

Currently, we do not cover those system services whose main logic and security checks are all performed in native code, for example, the Camera Service. These native services are small in number: only Camera Service, Media Player Service, Audio Policy Service, Audio Flinger and Sound Trigger Hardware Service were found in the Android 5.1 source code. While this may introduce false negatives, we believe the impact is minimal because most system services are implemented in the Java code.

3.3.3 Call Graph Construction

A precise call graph is the foundation of discovering inconsistent security policy enforcements in our approach. This phase computes the call graph for the entire Android framework. We rely on a context-insensitive call graph [94] which is light-weight and easier to build compared to a context-sensitive call graph, further discussion on the context-sensitivity may be found in S 3.6.

To construct the call graph, we need to know, for each call-site, all of its possible targets. As is common for object-oriented languages, the target of a method call depends on the dynamic type of the receiving object. A polymorphic method, *i.e.,* virtual method may have multiple implementations in descendant classes. The runtime has a dynamic dispatch

mechanism for identifying and invoking the correct implementation. Unfortunately, it is impossible for static analysis to collect runtime information and identify with 100% accuracy the callees of a virtual method. To address this problem, we use a conservative way to compute possible methods that might be called at a call-site. In other words, it computes an over-estimation of the set of calls that may occur at runtime using context information. Additionally, we connect IPC callers and callees directly to improve the precision and conciseness of the call graph. This is necessary because IPCs would introduce imprecision into our call graph. They use abstract methods to send data across process boundaries and thus in that procedure, many levels of virtual method calls are involved.

We take advantage of the solution proposed by PScout [57] to resolve Binder IPC calls and Message Handler IPCs. However, PScout fails to take a few special cases into account. For example, not all system services have an AIDL file that defines their remote interfaces. For instance, instead of using a stub class auto-generated from AIDL file, Activity Manager Service relies on a manually implemented class, `ActivityManagerNative`, to define its remote interfaces. Activity Manager Service extends `ActivityManagerNative` and implements these remote interfaces. Therefore, system services like Activity Manager Service should be handled carefully with additional logic.

Moreover, PScout does not consider another important IPC that is widely used by system services — Messengers and State Machines. System services expose AIDL methods that allow callers to obtain Messenger objects of their internal state machines. With a Messenger, an app or a system service can send messages to the corresponding state machine. Although in essence the communication between Messengers and State Machines is built on top of Message Handler IPCs, we find that PScout is unable to deal with this. We identify and connect all such senders and receivers for messages sent through Messenger objects.

Entry points. Similar to Java programs, the Android framework has a main method `SystemServer.main()`, from which system services are initialized and started. How-

ever, we cannot use it as the entry point. All service interfaces are likely to be called by a client app, but the construction and initialization procedure of system services cannot cover all remote interfaces. As a result, the framework call graph would be incomplete if we use `SystemServer.main()` as our analysis entry point.

Preprocessing produces a list of app-accessible service interfaces. Since we would like to include all of them in the call graph, one possible approach is to build call graphs from each of the interfaces, then combine these call graphs together to form the framework's call graph. This is not efficient because many call-sites would be included and computed multiple times. To cope with this problem, for each system service we construct a *dummy main* method, in which we construct the service object and enumerate all its app-accessible interfaces. The implementation details are described in §3.4.1.

3.3.4 Call Graph Annotation

This phase annotates the framework call graph with security check information. More specifically, given the types of security checks that are of interest to Kratos, Kratos automatically determines which security checks are performed by which call graph nodes, or methods, and annotates the nodes with security enforcement information, *e.g.,* permissions it enforces, UIDs it checks.

Identifying permission check methods. Android permissions are represented as string constants in the framework source code. When performing permission checks, a permission string is passed as an argument to a check method. According to developer comments in the Android source code, `checkPermission` in Activity Manager Service is the only public entry point for permission checking. Therefore, methods that eventually call `checkPermission` are considered as performing permission checks. We also want to know which particular permission is checked. To achieve this, we keep track of permission string constants passed to permission check methods. We observe that a few naming patterns can indicate whether a method is a permission check

method, such as `checkCallingPermission`, `enforceAccessPermission`, and `validatePermission`. Their names start with "enforce", "check" or "validate", and end with "Permission." Essentially, they are just wrappers of `checkPermission`, but we can leverage such patterns to make permission check method identification faster.

Identifying other security enforcements. Apart from permission checks, Kratos can also identify three other types of security checks automatically. For UID checks, they always get caller's UID using `Binder.getCallingUid()` and compare it with a constant integer. We use *def-use* analysis [135] to track which constant value the caller's UID is compared with. The Android framework reserves a list of UIDs and their values are defined in `android_filesystem_config.h`, from which we can identify the user that a given UID represents. Package name checks are more complicated. Besides the method shown in Figure 3.4 that uses a similar approach to permission checks, there also exist package name checks that are conducted like UID checks. Similarly, we employ def-use analysis and examine if the package name returned from Package Manager Service is compared with a string. In summary, to detect package name checks, we use both approaches for identifying permission checks and UID checks. Currently, there are no explicit hints that can instruct us to find a good way to identify thread status checks. Fortunately, their number is small, which allows us to identify them manually.

Annotating call graph nodes. After all security enforcement methods are identified, Kratos iterates call graph nodes and annotates them with security checks (labels) that are performed within. Labels are propagated toward the root of each sub-call graph with the union operator used to merge multiple labels at a node. This annotated call graph is then used in the next phase for detecting inconsistent security policy enforcement.

3.3.5 Inconsistency Detection

Inconsistency Detection consists of three phases. First, for every service interface Kratos obtains its sub-call graph from the framework-wide call graph and does forward

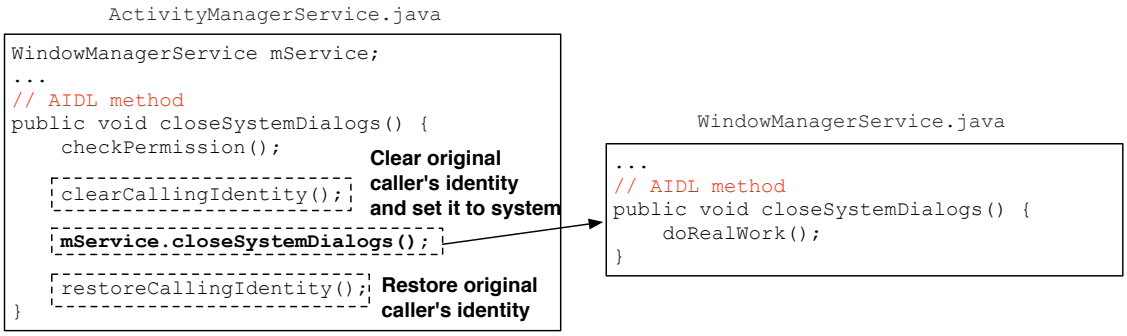

Figure 3.6: Activity Manager Service calls Window Manager Service to do the real work.

analysis on it to determine which security checks must be passed to reach each node. Next, Kratos compares service interfaces' call graphs in a pairwise fashion. Those pairs invoking the same method but with different security enforcements are considered as inconsistency candidates. Finally, Kratos applies three heuristics to rule out false positives.

With the annotated framework call graph, it is easy to obtain the sub-call graph of a service interface. For a sub-call graph, Kratos traverses all its nodes and summarizes the set of security enforcements required to reach each node from the root, by accumulating security enforcements along the path from the root to that node. Note that in system services `clearCallingIdentity()` is frequently used to clear the original caller's identity, or UID, and set caller identity to the system service temporarily. As Figure 3.6 depicts, after certain operations, the caller's identity is restored by calling `restoreCallingIdentity()`. If a method is called between `clearCallingIdentity()` and `restoreCallingIdentity`, all security checks are successfully passed because it appears as being called by the system service, which has elevated privileges. Thus, it is unnecessary for Kratos to perform any analysis between the two function calls.

After annotating sub-call graphs for service interfaces, we perform pairwise comparisons starting from their entry points to check if they ever invoke the same method, or converge on the same node. If such a convergence point exists for any two service interfaces, we believe they overlap in functionality and examine their paths that lead to the

convergence point. Specifically, we compare security enforcements along the two paths to see if they are consistent, *i.e.,* one path has a security check while the other does not.

3.3.5.1 Reducing False Positives

Not all methods are used to access system resources or perform sensitive operations. If we use arbitrary convergence between call graphs to indicate they have similar functionality, there would be a large number of false positives. For example, many methods are frequently called, such as `equal()`, `toString()`, `<init>()`, `<clinit>()`, but they are not sensitive and do not reflect the caller's functionality. To reduce the number of false positives, we investigate service interface call graphs, as well as the Android source, and design three heuristic rules. FigureWe observed that sensitive, low-level methods reside within the service side and are not accessible to apps. The runtime does not load them into an app's execution environment. Therefore, we can filter out methods that appear in an app's runtime. To achieve this, we classify classes imported by system services into three categories: (1) classes only used by system services, (2) classes used by both services and apps, and (3) classes only used by apps. Methods from the last two categories are believed to be insensitive and we discard them.

Second, we observe that many services have paired "accessor" and "mutator" methods, whose functionality is obviously different. For example, in Window Manager Service, `getAppOrientation` and `setAppOrientation` are used to get and set the app's orientation, respectively. Similarly, there exist other method pairs in which the two have opposite functionality, such as `addGpsStatusListener` and `removeGpsStatusListener`, `startBluetoothSco` and `stopBluetoothSco`. If such methods are found overlapping, we are confident that it is a false positive.

Third, we prioritize service interface pairs by calculating the sub-call graph similarity score of each pair. We also group together system services providing similar functionality, *e.g.,* the Telephony Service and the Telecom Service. Overlapping service interfaces belong

to the same group have higher priority to be manually examined. The rationale behind this is that if two services with no explicit connection (*e.g.,* the Power Manager Service and the SMS Service) are found overlapping, it is highly likely a false positive.

3.4 Implementation

We implement Kratos with around 15,000 lines of Java, Bash and Python. Based on the design described in §3.3, in this section we elaborate our implementation choices of Kratos. To ensure efficiency and scalability, we make an effort to parallelize the implementation. Our implementation the logic shown in Figure 3.5: (1) Preprocessing, (2) Call Graph Construction, (3) Call Graph Annotation, and (4)Inconsistency Detection.

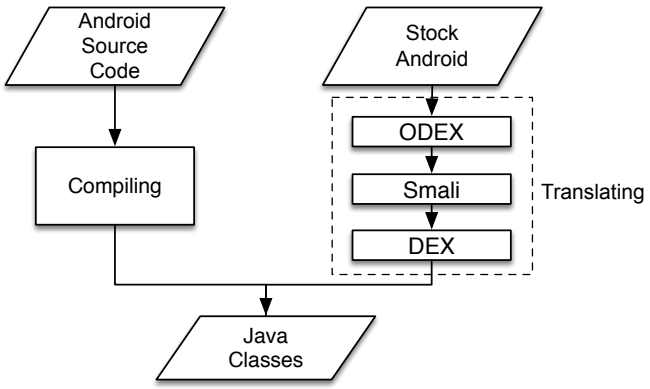

Figure 3.7: Getting Java classes from AOSP and customized frameworks.

3.4.1 Preprocessing

In the Preprocessing step, we obtain the necessary class files for the particular Android framework version. In the case of analyzing AOSP, it is a matter of compiling the Android operating system from the source code and extracting the class files.

For a vendor-specific version, it takes extra effort to obtain the class files. Because we do not have access to the source code of the customized framework, we must dump `odex` files from the device image, and translate them into corresponding Java `class` files.

Figure 3.7 shows the three steps involved in this process. We use *baksmali* [37] to convert `odex` into an intermediate format, `smali`. Then we employ *smali* [37] to assemble `smali` files into `dex`, and finally use *dex2jar* [43] to get JAR files, *i.e.,* Java classes. We notice that since Android 5.0 the Dalvik runtime has been replaced by the Android runtime (ART) [13], in which `odex` files are no longer available. To deal with that, *Dextra* [17] can be used to dump `dex`s from ART's `oat` files. Figure Once the class files are obtained, we utilize the Soot framework [136], a Java decompiler and analysis tool, to parse any given class and its member bodies, in order to identify which classes are app-accessible system services. More specifically, those services are identified by looking for invocations of `publishBinderService` and `addService`, two methods used for registering services to the global service manager. We exclude services registered by calling `publishLocalService`, as they are only available for system use. That means they are not accessible for third-party apps. We then distinguish app-accessible interfaces exposed by these app-accessible system services. For AIDL methods we look for their AIDL definitions, either in `aidl` files or in a public Java interface extending `IInterface`. Unprotected broadcast receivers can be identified by analyzing calls of `registerReceiver` and `registerReceiverAsUser`. If they do not have a `broadcastPermission` argument (or the argument is null) and intent actions the broadcast receiver listens to are not defined as `protected-broadcast` in the framework's `AndroidManifest.xml`, we consider this receiver as unprotected.

Once the identification of app-accessible system interfaces has finished, we take one last important step. We build an artificial single entry point for further analysis that uses Spark [97], a popular Java points-to analysis tool and call graph generator. It was designed to start at the program's single entry point, look for method calls there, then take all found callees, look at what callees call, and so on. This way, it builds a precise graph of what method is potentially called and identifies the methods which are reachable over all [31]. While the Android framework does have a static `main()` method in the System Server

class, there is no guarantee that all methods will be called from that point of origin, as that is only responsible for instantiating system services. Thus, we must provide Spark a single static entry point into the Android framework for each class analyzed.

We use method and class instrumentation data structures provided by Soot to dynamically build "wrapper" classes with a static main method. These wrappers are a necessity to meet Spark's requirement of static entry points for invoking method calls of a class. Kratos automatically builds the wrapper classes by inferring important attributes of service interfaces. Class access modifiers are one key piece of analysis. Once they are understood by Kratos, it decides how to build a wrapper. In the best case, the service is a public class; while in the worst case, the service is a private inner class.

3.4.2 Call Graph Construction

In this phase, we utilize Spark to generate a context-insensitive call graph that encompasses all app-accessible service interfaces. We use the dummy main method as the single entry point. For Spark to generate the call graph, it must operate on one thread; thus we are unable to parallelize this phase. It is in this phase that Java virtual method resolution occurs, by leveraging variable-type analysis (VTA). We also enable the *on-the-fly* option, because it was reported that the most effective call graph construction method proceeds on-the-fly and builds the call graph at the same time as it computes points-to set [103].

3.4.3 Inconsistency Detection

In order to discover inconsistencies within security enforcements, we intelligently match two call graphs of different service interfaces in a pairwise fashion. Because this phase can run, at worst, in $O(n^2)$ time, we use a set of heuristics in order to reduce the total number of comparisons, which is outlined in S 3.3. Once two call graphs are paired, we look for a point of convergence — a method whereby both paths will intersect. Once we find an intersection, we use backward analysis to identify any other services that can reach

the method. This allows Kratos to quickly identify additional services which may share that path in which security circumventions occur.

To prioritize the results for manual validation, we use fast belief propagation to measure node affinity, and then calculate the sub-call graph similarity score with the Matusita distance. Denote the final affinity score matrix as S, we have $S = [I + \epsilon^2 D - \epsilon A]^{-1}$ where I is the identity matrix, D is the degree matrix, A is the adjacency matrix, and ϵ is a small number which we take the value of 0.02. Matusita distance d is defined as $d = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}(\sqrt{S_{1,ij}} - \sqrt{S_{2,ij}})}$ where $S_{1,ij}$ and $S_{2,ij}$ are entries of S for the sub-graphs. And the similarity score $sim = \frac{1}{1+d}$ is then calculated.

3.5 Results

Table 3.1: Results of the six codebases in our evaluation. We only consider services implemented in Java.

Codebase	**# Services**	**# Service Interfaces**		**# Class Files**
		# AIDL Methods	# Broadcast Receivers	
Android 4.4	70	1,010	26	14,901
Android 5.0	89	1,483	28	33,110
Android 5.1	89	1,510	31	33,433
Android M Preview	89	1,490	31	35,431
AT&T HTC One (Android 4.4.2)	85	1,868	35	17,879
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	159	2,463	64	171,306

In this section, we evaluate Kratos' effectiveness, accuracy, and efficiency by applying it to six different Android frameworks. We also present vulnerabilities identified using Kratos and analyze some of them in detail. All our experiments are conducted on a desktop machine with a 3.60GHz 8-core Intel Core i7 CPU and 16GB memory, running 64-bit Ubuntu Linux 14.04.

Codebases. We target both AOSP Android and customized Android. Since the Android framework is evolving, in addition to inconsistencies within a particular Android framework codebase, we also track inconsistencies across different Android versions. Therefore,

we choose four releases, *i.e.,* Android 4.4, 5.0, 5.1, and 6.0. Vendors and carriers often change existing or add new code to provide a unique and differing experience. Previous work [143, 92] reported security threats brought by such customizations. We believe that they may also lead to more inconsistencies as different parties are involved, and their engineers are likely to have a different understanding of the security policy. Specifically, we analyze two customized Android frameworks, AT&T HTC One and T-Mobile Samsung Galaxy Note 3, both based on Android 4.4.2.

Table 3.1 summarizes the statistics of the six Android framework codebases in our evaluation. For the AOSP Android, the number of services increases dramatically from version 4.4 to 5.0, then remains unchanged in 5.1 and M preview. However, as the second column shows, the number of AIDL methods exposed by system services drops by 20 in M preview. It is obvious that Samsung and T-Mobile customize Android much more heavily than HTC and AT&T (mostly contributed by Samsung). Though both phones are based on the same version of AOSP codebase, Galaxy Note 3 has 89 more system services while HTC One only has 9 more. Moreover, customization also increases the number of service interfaces, as well as class files.

Tool Efficiency. We measure Kratos's efficiency and summarize the results in Table 3.2. The Preprocessing phase only takes a few minutes. Call Graph Construction and Call Graph Annotation are very fast, each finishing within one minute. Even though Inconsistency Detection consumes the majority of processing time, we can analyze a framework codebase in less than 20 minutes.

3.5.1 Tool Effectiveness

Table 3.3 summarizes our overall analysis and detection results on all six Android framework codebases. The first column is the number of inconsistent security policy enforcement Kratos discovered. To evaluate true positive (TP) and false positive (FP), we manually examine all cases of enforcement inconsistency. The numbers of true positives

Table 3.2: Time consumed in each analysis step of Kratos (in seconds)

Codebase	Preprocessing	CG Construction	CG Annotation	Detection
Android 4.4	95.4	23.4	8.6	470.3
Android 5.0	137.1	25.0	10.53	496.4
Android 5.1	209.0	22.2	14.6	445.9
Android M Preview	141.6	21.6	9.7	482.3
AT&T HTC One (Android 4.4.2)	110.8	29.1	16.0	655.8
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	306.9	57.5	50.7	1273.7

Table 3.3: Overall results of Kratos. The numbers of exploitable inconsistencies, true positives and false positives are concluded by manual analysis.

Codebase	# Inconsistencies	# TP	# FP	Precision	# Exploitable
Android 4.4	21	16	5	76.2%	8
Android 5.0	61	50	11	82.0%	11
Android 5.1	63	49	14	77.8%	10
Android M	73	58	15	79.5%	8
AT&T HTC One (Android 4.4.2)	29	20	9	69.0%	8
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	128	102	26	79.7%	10

and false positives are listed in column 2 and column 3, respectively. We also manually validate exploitable inconsistencies for each codebase, and show the results in the last column. We consider inconsistent enforcement cases which can be exploited by a third-party application as exploitable, as they are more likely to result in real-world attacks.

For the four AOSP codebases, Kratos reports more inconsistencies in newer versions. There are only 21 inconsistencies in Android 4.4 framework. However, this number drastically increases to 61 in the later version, Android 5.0. This is to be expected, as shown in Table 3.3, Android 5.0 introduces 19 more system services. More interestingly, many of the new system services seem to have similar functionality to existing ones. For example, the RTT (round trip time) Service introduced in Android 5.0 can be used to measure the round trip time of accessible Wi-Fi access points nearby. Incidentally, the Wi-Fi Service also provides similar functionality. Another example is Telecom Service, whose functionality overlaps with Telephony Service. Meanwhile, the large number of system services

added by T-Mobile and Samsung undoubtedly introduce more inconsistencies.

True positive and false positive. For all codebases except the one from HTC One, Kratos can achieve more than 75% precision. We cannot measure the false positive rate because we do not have other sources of data with ground truth of known inconsistencies. Therefore, it is not feasible for us to calculate the number of true negatives and false negatives. We further analyze false positive cases and try to understand why they occur. We find that most false positives are caused by the three limitations of Kratos. First, two service interfaces are not equivalent in functionality, yet they invoke the same underlying sensitive method, which is invoked with different arguments. Since Kratos uses path-insensitive analysis, it cannot discern the impact differing arguments have on the execution path. For example, Account Manager Service has two public interfaces: `getAccounts()` and `getAccountsForPackage()`. The former can list all accounts of any type registered on the device, while the latter returns the list of accounts that the calling packages are authorized to use. They eventually call `getAccountsAsUser()` with different arguments, and `getAccountsForPackage()` has one more security check — a UID check which ensures that the caller is an authorized user.

The second limitation is the inaccuracy of the overlapping service interfaces reported by Kratos. As we have mentioned in §3.3, the over-estimated call graph could introduce false positives. Spark utilizes point-to analysis, which makes every effort to resolve virtual method calls according to context. Nevertheless, it cannot resolve all virtual methods with 100% accuracy.

The third limitation also comes from service interfaces with similar but not equivalent functionality. One might be more capable than the other one, and the service with more capability is guarded by stricter security enforcement. For example, `deleteHost()` and `deleteAllHosts()` from App Widget Service are able to delete host records. They both call `deleteHostLocked()`. The difference is `deleteHost()` calls `deleteHostLocked()` only once, while `deleteAllHosts()` calls it multiple times

in a loop. The latter appears to be more powerful, as it can delete all host records while the former can only delete one record per call. Compared to `deteleHost()`, `deleteAllHosts()` checks a caller's UID. Kratos is not able to recognize method body semantics in order to evaluate a service interface's capability.

Not all inconsistencies are exploitable. Note that among all true inconsistency cases, only a small portion of them (18.3%) are exploitable by a third party. The reason is threefold. First, they may both require system-level permissions. Our attack model assumes that an attacker builds a third-party application and manages to have it installed on a victim's Android device. While it is possible for the methods to be invoked, system-level permissions are inaccessible by third-party applications. Two services in the "HTC One" code base provide telephony functionality, *i.e.,* HtcTelephony Service and HtcTelephonyInternal Service. They both expose an interface `setUserDataEnabled()` for enabling and disabling cellular data connection, and both invoke `Phone.setUserDataEanbled()` to finish the request. Kratos reports that permissions used in the enforcement are different. HtcTelephony only enforces `APP_SHARED`, but HtcTelephonyInternal enforces `APP_SHARED` together with `CHANGE_PHONE_STATE`. We cannot exploit this inconsistency, because `APP_SHARED` is a system-level permission.

The second reason that an identified inconsistency is not exploitable stems from the difficulty to construct valid arguments for calling a service interface without a required permission, even though the interface has a weaker security enforcement than another service interface providing the same functionality. For instance, Connectivity Service exposes `isActiveNetworkMetered()` and Network Policy Management Service defines `isNetworkMetered(NetworkState)` to allow callers to query if the active network is metered. Kratos reports that `isActiveNetworkMetered()` enforces a permission `ACCESS_NETWORK_STATE`, but `isNetworkMetered(NetworkState)` does not. This is a true inconsistency. However, to invoke `isNetworkMetered(NetworkState)`, one must obtain or in-

stantiate a `NetworkState` object, which requires the `ACCESS_NETWORK_STATE` permission. In the end, the same permission is required in order to successfully invoke these two interfaces.

Third, the existence of feature checking logics makes it difficult to reach to particular methods. Some resources could be accessed only when a certain feature is satisfied. Sometimes, a security checking function is not directly called, and instead, an object (could either be a flag or instance of another class) is verified where the object itself will only be valid if a security check is passed.

Characteristics of the vulnerabilities. Interestingly, we find many vulnerabilities are discovered only when we analyze hidden interfaces. In fact, 11 of them are exploitable through hidden interfaces that are not directly visible to applications. Theoretically, these hidden interfaces are not expected to be used by developers, but Android does not restrict applications to access them through Java reflection. This finding suggests that hidden interfaces are not carefully scrutinized. Perhaps disabling reflection would be one way to reduce such attack surface. In addition, we find three vulnerabilities are discovered by analyzing two different services which performed the same sensitive operation, which shows that functionalities are sometimes redundant across services. Besides, we find four vulnerabilities where a system permission is bypassed, allowing a third-party application to perform operations that are absolutely disallowed by Android.

In summary, these results demonstrate that although human efforts are indispensable, Kratos is effective at automatically detecting a variety of inconsistent security enforcement. Based on the cases identified, Kratos is able to uncover previously unknown vulnerabilities.

3.5.2 Case Studies

By analyzing security enforcement inconsistencies reported by Kratos, we have discovered 14 vulnerabilities, summarized in Table 3.4. N/A means the vulnerability only exists in customized Android and does not affect other frameworks.

Table 3.4: Summary of inconsistent security enforcement that can lead to security policy violations.

Service[1]	Affected Framework						Description	Attack	Bypassed Security Enforcement[2]
	AT&T HTC	T-Mobile Samsung	4.4	5.0	5.1	M Preview			
SMS	✓	✓	✓	✓	✓	✗	Clear all SMS notifications showing in the status bar	Privilege escalation	Package Name (SMS)
Wi-Fi	✓	✓	✓	✗	✗	✗	Set up an HTTP proxy that works in PAC mode	Privilege escalation	CONNECTIVITY_INTERNAL*
NSD	✓	✓	✓	✓	✓	✓	Enable/Disable mDNS daemon with only INTERNET permission	DoS	CONNECTIVITY_INTERNAL*
RTT	✗	✗	✗	✓	✓	✓	Crash the Android runtime	Soft reboot	ACCESS_WIFI_STATE
Wi-Fi Scanning	✗	✗	✗	✓	✓	✓	Crash the Android runtime	Soft reboot	ACCESS_WIFI_STATE
GPS	✓	✓	✓	✓	✓	✗	(1) Send raw data to GPS's native interface (2) Crash the Android runtime	Privilege escalation, Soft reboot	ACCESS_FINE_LOCATION
GPS	✓	✓	✓	✓	✓	✓	Get GPS providers that meet given criteria	Privilege escalation	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION
Input Method Management	✓	✓	✓	✓	✓	✓	Dismiss input method selection dialog	DoS	UID (SYSTEM)
Telephony/ Telecom†	✗	✗	✗	✓	✓	✓	End phone calls without any permissions	Privilege escalation	MODIFY_PHONE_STATE* CALL_PHONE
Telecom	✗	✗	✗	✓	✓	✓	Get phone state without any permissions	Privilege escalation	READ_PHONE_STATE
Activity Manager/ Window Manager†	✓	✓	✓	✓	✓	✓	Close system dialogs	DoS	UID (SYSTEM)
Power Manager/ Persona Manager†	✓	✓	✓	✓	✗	✗	Set maximum screen timeout	Draining battery	UID (ADMIN, SYSTEM)
Device Info	N/A	✓	N/A	N/A	N/A	N/A	Save MMS to audit database	Privilege escalation	UID (PHONE)
Phone Interface Manager Ext	N/A	✓	N/A	N/A	N/A	N/A	Send raw request to radio interface layer (RIL)	Not clear	MODIFY_PHONE_STATE*

[1] Items in this column labeled with † indicate that the inconsistency occurs between two services.
[2] Permissions labeled with ∗ are system permissions that cannot be used by third-party applications. Specific UIDs that can be bypassed are parenthesized.

We have filed 8 security reports regarding these vulnerabilities to the Android security team. All of the vulnerabilities we reported have been acknowledged and confirmed. Among them, the mDNS daemon vulnerability was originally classified as a high severity vulnerability, but then rated as low severity. The ones in Wi-Fi Service and Power Manager Service had been fixed before we reported it. Note that we are very conservative about the results, which means those confirmed in the code but have not been validated in real devices are not counted. In this section we select several vulnerabilities shown in Table 3.4 and explain them in detail.

Starting/Terminating mDNS daemon (denial of service). The multicast Domain Name System (mDNS) provides the ability to perform DNS-like operations on the local area network in the absence of any conventional Unicast DNS server [25]. Android starts an mDNS daemon `mdnsd` when the system boots up. This daemon is used and controlled by the Network Service Discovery (NSD) service, which allows an application to identify other devices on the local network that support the services it requests [45]. It is useful for a variety of peer-to-peer applications such as file sharing and multiplayer gaming. NSD Service exposes an interface that is able to start and terminate `mdnsd`. Considering the importance of the mDNS daemon, that interface is protected by a system-level permission, `CONNECTIVITY_INTERNAL`, which cannot be acquired by third-party applications. Therefore, by design, all attempts made by third-party applications to start or terminate the mDNS daemon is thwarted.

However, another interface exposed by the NSD Service, `getMessenger()`, could be used to achieve the exact same functionality. By calling `getMessenger()`, the caller obtains a reference of the Messenger object from NsdStateMachine that manages the communication with `mdnsd`, then can send messages to it. Compared to the interface protected by the `CONNECTIVITY_INTERNAL` permission, this interface only checks the `INTERNET` permission, which is one of the most frequently requested permissions [142]. Considering that the `INTERNET` permission is so commonly used and low-privilege, applications that request it do not raise a user's attention. NsdStateMachine distinguishes different types of incoming messages by examining their `what` field. A malicious application with only `INTERNET` permission can easily craft a message, set its `what` field to `NsdManager.DISABLE`, and send it to NsdStateMachine, to terminate the mDNS daemon. As a result, users can no longer use applications that rely on the NSD Service.

Ending phone calls (privilege escalation). Both Telephony Service and Telecom Service provide a method `endCall()` that allows caller applications to reject incoming phone calls and end ongoing phone calls. According to Android's

source code, their corresponding wrappers, `TelephonyManager.endCall()` and `TelecomManager.endCall()`, are annotated with `@hide` and `@SystemApi`, respectively. That means both should only be used by the system. In fact, Telecom Service's `endCall()` indeed enforces a system permission, `MODIFY_PHONE_STATE`, ensuring that only the system is able to use it. However, Telephony Service's `endCall()` only checks the `CALL_PHONE` permission, which can be acquired by third-party applications. More interestingly, a component of Telephony Service registers a broadcast receiver in which phone calls are ended when a specific broadcast comes in. Unfortunately, this broadcast receiver is not protected at all, making it possible for an application without any permissions to end phone calls. This inconsistent security enforcement allows an attacker to create denial of service attacks against compromised devices. It could also be exploited by ransomware to make victims' phones unusable.

Dismissing SMS notifications (privilege escalation). A `MessagingNotification` object instantiated in the SMS Service registers a broadcast receiver to listen to message deleting broadcast. If such a broadcast is received, it clears all SMS notifications showing in the status bar. Kratos reports that this receiver is not protected by any permission, and it has similar functionality to the Notification Service. By design, notifications can only be dismissed by their owners or the system, enforced by a package name check. However, we can bypass this enforcement by sending the broadcast to the SMS Service. This bug only affects Android 5.1 and earlier versions because "the MMS app no longer ships with latest versions of the OS."

Crashing the Android runtime (soft reboot). The Wi-Fi Scanning Service provides a way to scan the Wi-Fi universe around the device. Similar functionality is provided by the Wi-Fi Service as well. They both leverage the `WifiNative` class that is responsible for communicating with the native binary `wpa_supplicant`, from which Wi-Fi scanning results can be obtained. Kratos reports that Wi-Fi Service checks `ACCESS_WIFI_STATE` permission, while Wi-Fi Scanning Service has no permission enforcement. We attempt to

exploit this inconsistency to query Wi-Fi scanning reports without declaring any permissions. It turns out due to implementation issues of the Wi-Fi Scanning Service, it causes a crash of the entire Android runtime.

We further analyze the source code and crash log. In fact, we could successfully trigger the invocation of `WifiNative.startScanNative`. Since we have to stop the scanning before reading the results, we attempt to call `WifiNative.stopScanNative`, in which a runtime exception is thrown out at line 65 of `art/runtime/check_jni.cc`. The exception is not handled, therefore the runtime crashes, causing a soft reboot.

Setting maximum screen timeout (draining battery). In T-Mobile Samsung Galaxy Note 3, Kratos finds two service interfaces with exactly the same name but different security enforcement for calling `setMaximumScreenOffTimeoutFromDeviceAdmin()`. One is exposed by the Power Manager Service from the AOSP codebase based on which customization was made; another is exposed by the Persona Manager Service from the customized portion. These two methods implement the same functionality but their security enforcement is inconsistent. The Power Manager Service does not apply any security checks on its `setMaximumScreenOffTimeoutFromDeviceAdmin()`, however, Persona Manager Service checks the caller's UID.

The name of the method implies that it should only be used by the device administrator, but Kratos did not find any checks. We further analyze the AOSP source code and confirm that it is a real inconsistency in security enforcement. In the comments, the developer made it very clear that this method should only be called by a device administrator (as shown in left side of Figure 3.1). But surprisingly, they did not apply any security checks to secure it. By invoking Power Manager Service's `setMaximumScreenOffTimeoutFromDeviceAdmin()`, an application without the proper permissions can set the screen timeout to a very large value in order to drain the battery. Past studies [146, 69, 130] have shown that display is a major contributor to the battery consumption of smartphone users.

In this case, the inconsistency occurs between two codebases — the original AOSP code and customization code. This case also demonstrates that though customization is often blamed for introducing more security threats, it is also possible that customizations are more secure than AOSP.

Sending raw requests to RIL. We found two system services in the Samsung Galaxy Note 3 that provide very similar telephony-related functionality. These two services are implemented in two classes, `PhoneInterfaceManager` and `PhoneInterfaceManagerExt`. Kratos reports that the application-accessible interface `invokeOemRilRequestRaw()` exposed by PhoneInterfaceManager and another interface `sendRequestRawToRIL()` exposed by PhoneInterfaceManagerExt mirror in functionality. Specifically, they both invoke `Phone.invokeOemRilRequestRaw()` to send raw requests to radio link layer (RIL). Nevertheless, their security enforcement is different. `invokeOemRilRequestRaw()` checks the `CALL_PHONE` permission, while `sendRequestRawToRIL()` has no security checks. As a result, using `sendRequestRawToRIL()`, an attacker can send arbitrary data to RIL without requesting any permissions. Note that attackers can only control data, but cannot alter the request type. `sendRequestRawToRIL()` restricts request types to `RIL_REQUEST_OEM_HOOK_RAW`. We monitor all RIL requests sent by a test-phone and confirm that this request type is used. We have not managed to craft malicious data to take control of RIL or attack base stations, because of the difficulty involved in reverse engineering the protocol. However, we believe this unprotected service interface can be exploited by sophisticated attackers who have more knowledge of cellular networks, especially the use of `RIL_REQUEST_OEM_HOOK_RAW` request.

Our further investigation of AOSP Android 5.0 source code reveals that the method `invokeOemRilRequestRaw()` is actually protected by a system permission `MODIFY_PHONE_STATE`, which is higher-privileged than `CALL_PHONE` that this T-Mobile Samsung phone's `invokeOemRilRequestRaw()` enforces. This implies that

vendors/carries and Google engineers do have a different understanding of how to protect certain sensitive operations.

3.6 Discussion and Limitations

False negatives. Similar to previous static analysis work, our approach can miss security enforcement inconsistencies. First, Kratos is not able to deal with implicit control flow transitions, *e.g.,* callbacks. To address this problem, we could implement some principles found in EdgeMiner [68] and FlowDroid [56]. Namely, their implicit control flow and context/flow/lifecycle analysis principles, respectively. While this would help identify another security check present within the Android framework, it doesn't completely solve our false negative problem.

Because we use heuristics to reduce computation time for identifying security enforcement circumventions, there is the potential that a heuristic may rule out a true positive. As with all heuristics, they come at the cost of introducing false negatives or false positives. This very tradeoff is one we work hard to balance; keep the runtime fast and minimize the false positive and false negatives. Moreover, vendors/carriers may introduce new means for enforcing their security policies besides the four we have considered.

Kratos currently does not handle the native code (*i.e.,* binaries compiled from C/C++ code) that comprises the lower levels of Android and some small portion of the service codebase, leading to false negatives. To address this, additional work is needed to build and analyze a control flow graph at the native layer, which is part of our future work.

Difficulty in verifying violations. Currently, Kratos is unable to automatically verify the presence of a circumvention. To decide if a violation is exploitable, one needs to understand the semantics of the code. In most cases, a review of the actual source code is the only way to ascertain the semantics. One must know what operations are able to interact with untrusted space and also design a feasible way to mount the attack. This is the most time-consuming portion of Kratos.

Context-insensitive and path-insensitive analysis. Kratos depends on Spark to build a context-insensitive call graph, which could introduce false negatives or positives. It is understood, through work conducted by Lhoták and Hendren [106] , that a context-insensitive call graph may impact the call graphs accuracy. However, Lhoták and Hendren go on to prove that the improvements context-sensitivity provide to the accuracy are minimal. Through these findings, we justify our optimization for a context-insensitive call graph; the memory and computational overhead of a context-aware call graph analysis does not improve the accuracy enough relative to its costs. Path-insensitivity is not applicable here because we are not interested in how branching affects a call chain. We are only interested in security enforcement circumvention, which does not depend on branching. Thus, we are able to safely ignore utilizing this analysis for our call graph.

Scalability. While we have made every effort to allow Kratos to be scalable at every facet, there is one specific place in which we cannot run a parallelized computation — our use of Spark. Spark has significant limitations in scalability. Because of the reliance on Spark, the "Call Graph Construction" phase is unable to be threaded. Even in spite of this limitation, the 3.4.2 phase completes in minimal time (see results in Table 3.2). Every other aspect of Kratos leverages threading in an effort to reduce run time.

Native system interfaces. We also observe an interesting case where an app can get the MAC address of a network interface card (NIC) using three different ways, guarded by different security enforcement. The first approach is to call the Connectivity Service's `getLinkProperties()`, which checks `ACCESS_NETWORK_STATE` permission. Second, an app can run the command line tool `/system/bin/netcfg` to obtain a list of available NICs and information, including MAC addresses. This requires the app owns `INTERNET` permission. However, the third approach, reading MAC address directly from the file `/sys/class/net/[nic]/address`, does not require any permissions. This motivates our future improvement of Kratos—to handle inconsistencies across different layers of Android.

3.7 Summary

In this work, we propose Kratos, a static analysis tool for systematic discovering inconsistencies in security enforcement which can lead to security enforcement circumvention vulnerabilities within the Android framework. We have demonstrated the effectiveness of our approach by applying our tool to four versions of Android AOSP frameworks as well as two customized Android versions, conservatively uncovering at least 14 highly-exploitable vulnerabilities that can lead to security and privacy breaches such as crashing the entire Android runtime, arbitrarily ending phone calls, and setting up an HTTP proxy with no permissions or only low-privileged permissions. Interestingly, many of these identified inconsistencies are caused by the use of hidden interfaces of system services. Our findings suggest that some potentially promising directions to proactively prevent such security enforcement inconsistencies include reducing service interfaces and restricting the use of Java reflection (for accessing hidden interfaces).

We have shown that security enforcement circumvention is a systemic problem in Android. Our work demonstrates the benefit of an automatic tool to systematically discover anomalies for security enforcement in large codebases such as Android. We expect Kratos to be useful for both Android developers as well as vendors who offer customized Android codebases.

CHAPTER IV

A Lightweight Framework for Fine-Grained Control of Application Lifecycle

This chapter presents our study on diehard applications that abuse system service APIs and lifecycle entry points. To fundamentally improve the design of Android's application lifecycle control, we propose a lightweight, fine-grained framework. Users, developers, and system designer can all benefit from our framework. Developers can utilize the framework to realize effective and efficient restrictions on diehard applications; users will get better experiences. System designers can integrate our framework to detect and restrict diehard applications, and the findings of this work are helpful with future API design.

4.1 Introduction

Mobile app lifecycle is dynamically managed by the system and is opaque to users. Due to the constrained resources on mobile devices, the Android system controls each app's lifecycle based on their demands and task priorities. In particular, Android imposes looser restrictions on app lifecycle and allows background execution without user awareness. On the one hand, Android's permissive lifecycle control gives apps more flexibility to react to user interactions and system events timely, and thus enables rich functionalities, such as background video recording. On the other hand, however, it also opens doors for apps to

directly or indirectly alter their lifecycles. In fact, apps can easily *abuse* their entry points to automatically start up in the background, requiring no user interaction, and *game* the lifecycle management mechanism to evade being killed.

We call app behaviors that make changes to their lifecycles for the purpose of 1) keeping long-running in the background or 2) evading being killed *diehard behaviors*. Apps exhibiting such diehard behaviors are thus *diehard apps*. Diehard apps can cause battery drain and device performance degradation. Since they are oftentimes completely invisible while running in the background, it is hard for normal users to be aware of their existence and what they are actually doing. It is reported that the Amazon Shopping app operates in the background so that it remains up to date with current offers and promotions, causing high battery usage [42]. People also have privacy concerns on such apps, as they could stealthily and constantly collect sensitive user data, such as geolocations [21, 30, 140].

```
1  // full class name: com.android.Laucher.Se
2  public class Se extends Service {
3    ...
4    // onDestroy() callback is always called by
5    // the system when a service gets killed
6    public void onDestroy() {
7      super.onDestroy();
8      ...
9      // Restart itself (the 2nd argument is the
10     // target service that will be started).
11     Intent i = new Intent(this.context, Se.class);
12     i.setFlags(268435456);
13     i.setAction("com.dai.action");
14     i.setAction("com.tdz.action");
15     this.startService(i);
16   }
17   ...
18 }
```

Figure 4.1: Code snippet of the HummingBad malware, decompiled by JEB Decompiler. The target of the intent object (local variable `i`) is set to `Se.class`, meaning that the service attempts to restart itself while being killed.

Essentially, diehard apps exploit two fundamental problems in Android app lifecycle.

Table 4.1: The changes cause lifecycle fragmentation, *i.e.,* an app's lifecycle is inconsistent in different Android frameworks.

Android version	Improvements	Diehard techniques affected
Marshmallow (6.0)	Doze, App Standby	Alarm Manager, Long-lived TCP connections
Nougat (7.0)	Fixing notification bug, Doze on the go, Background Optimization	Hiding notifications
Oreo (8.0)	Job scheduler improvements, Background Execution Limitation	Static broadcast receivers
Pie (9.0)	Background Restrictions	Foreground services

First, apps can have multiple entry points that are by default accessible to other apps on the same device. In addition to the user starting an app explicitly, the app can be launched by the system or another app as well, requiring no user involvement. For example, the system broadcasts signal strength changes so that apps potentially affected by weak signal strength can take actions accordingly. A diehard app, however, can also claim to handle the event and it will be automatically launched by the system to process signal changes. Second, and more importantly, app lifecycle is not strictly enforced and is hard to enforce. The lifetime of an app process is determined by the system through a combination of the parts of the app that the system knows are running, how important these things are to the user, and how much overall memory is available in the system [32]. Since apps are a sophisticated interplay between custom code and the system framework, they are able to game the system to indirectly manipulate their own lifecycle states. For instance, apps with foreground services are believed to have higher priorities. Knowing this, diehard apps usually start foreground services to escalate their priorities even though it is not a necessary functionality for them. Moreover, apps can directly alter their component lifecycle. App components implement a series of callbacks which are invoked by the system through its lifetime, but there is no limitation on what they can do inside each callback. Malicious apps have been exploiting the loosely enforced app lifecycle to be diehard, *e.g.,* the notorious HummingBad malware. As Figure 4.1 shows, when its service gets killed, it attempts to restart the service immediately. Not all developers are well educated or are willing to

follow the guidelines. They get things done in ways they see fit, sometimes causing diehard behaviors unintentionally.

New but ad hoc features (summarized in Table 4.1) have been introduced to Android in an effort to limit background apps, for example, background optimization [14], Doze, and App Standby [28]. They affect diehard behaviors to a certain extent, but unfortunately, they cannot fundamentally solve the diehard behavior problem and they all have obvious limitations. First, there are legitimate cases where apps need to keep running in the background, but Background Optimization and Background Execution Limitation are both too coarse-grained, either allowing or disallowing *all* background activities. It is difficult to balance the trade-off between app functionality and user experience. For end users, neither zero control or excessively strict control is helpful. Second, apps can always find "creative" ways to bypass background restrictions. Diehard techniques evolve along with the Android framework. For example, developers have come up with approaches to escalating process priority so that their apps will not be killed when available memory is low. Malware variants use social engineering to bypass a battery-saving process and stay active in the background.

We acknowledge that certain apps may have legitimate reasons for being diehard. However, they should comply with system regulations and development guidelines for providing the best user experience. We argue that diehard behaviors violate the system's app lifecycle control and they should be better managed. Comprehensive modeling of app lifecycle which can enable fine-grained lifecycle control is desired.

In view of this need, we make the first effort towards providing fine-grained control of app lifecycle. In particular, we categorize diehard techniques that are used by apps to keep long-running, from which we learn a valuable insight that diehard apps create high-priority app components and/or develop interdependence between component callbacks, between app components, or between other apps. The complicated app lifecycle makes it challenging to realize reliable, systematic detection and restriction. To tackle this, we pro-

pose *Application Lifecycle Graph (ALG)*, a systematic, informative, and precise description of app lifecycle. The problem of diehard behavior detection thus can be transformed into operations on a directed graph, *i.e.,* the ALG. Specifically, the interdependence created by diehard apps can be identified as cycles, and diehard behaviors are reflected on the ALG as edges with special properties. Leveraging ALG, we develop a lightweight framework that enables flexible and fine-grained app lifecycle enforcement at runtime. We collect and analyze 17,598 apps from Google Play and a third-party app market. Results show that diehard behaviors are very common among apps. To our surprise, diehard behaviors sometimes come from third-party libraries an app integrates, making the host app a diehard parasite.

In summary, this work makes the following contributions:

- We propose app lifecycle graph (ALG), a fine-grained, precise description of system-wide app lifecycle. ALG allows us to transform diehard behavior detection and restriction problems into classic graph problems, *i.e.,* cycle detection and edge pruning.

- Leveraging ALG, we design and implement a lightweight runtime framework for fine-grained control of app lifecycle. This framework exposes a set of easy-to-use APIs and therefore enables the development of new functionalities in app lifecycle management.

- We perform the first study on diehard apps and diehard behaviors in the wild. We find that diehard behaviors are common among apps from both Google Play and a third-party app market. Anf interesting observation is that app developers may not intentionally make their apps diehard, but the third-party libraries they integrate have diehard behaviors.

4.2 Motivation

In this section, we identify the limitations of Android's application lifecycle management. We use real-world examples to demonstrate those limitations that motivate our work.

4.2.1 Component Lifecycle

Components are the essential building blocks of Android apps. There are four types of components that can be used within an app, *i.e.,* Activity, Service, Broadcast Receiver, and Content Provider. Each type of component has its distinct lifecycle. A component transitions through different lifecycle states during an app's execution and the framework calls its lifecycle callbacks [12] at each state change. It is the developers' responsibility to define how a component behaves in response to lifecycle state changes. For example, while a Service is being created, its `onCreate()` is called. Developers override the default callbacks, but there is no restriction on what they can do in each lifecycle callback. An app's lifecycle is far more complicated than the aggregation of all its components' lifecycles, because there exist control flows and data flows among components.

Inter-component communications (ICCs) occur both within individual apps and between different apps. ICC relies primarily on the exchange of asynchronous messages called *Intents*, which can carry extra data in the form of key-value pairs. The ICC initiator creates an Intent instance and puts into it the target component information. ICCs enable complicated collaborations across apps. For instance, the camera app allows users to share photos on social media conveniently, by sending an Intent object with photo information to the social app. If the target app is not running, the system starts it so that desired operations can be completed. Because of this design, an app can *wake up* other apps through ICC.

4.2.2 Memory Management

By design, Android does not immediately kill app processes when they are switched to the background. They are cached in the background so that they can be quickly recov-

ered when the user switches back. In this way, the system can speed up reopening apps if needed, but it also easily gets into a low free memory state, where it has to shut down certain processes in order to provide memory to processes that are more immediately serving the user [33]. If an app process gets killed, all components residing in that process are consequently destroyed. When deciding which processes to kill, the system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities. The decision of whether to terminate a process, therefore, depends on the states of the components in that process. The system uses Activity Manager Service to track the importance of processes and reflect their importance by setting the `oom_adj` (latest kernels use `oom_score_adj`) value of the process under `/proc/PID/`. The higher the `oom_adj` value is, the more likely this process gets selected by the kernel's low memory killer (LMK). Since a process may host multiple components at a time and priorities are per process, the state of one single component can affect the entire process's priority.

App components have their individual lifecycles, but the call graph of callbacks is incapable of describing an app's complete lifecycle. First, in addition to component callbacks, frequent ICCs are also part of app lifecycle. Second, an app can wake up another app when inter-process communications (IPC) occur. In fact, *fragmentation* aggravates the problem, as not all devices can be upgraded to the latest version. As of September 15th, 2018, 13 months after Android Oreo was released, 85.4% devices are still running older versions [18]. Device vendors such as Huawei customize Android and add their own power-saving features. However, they impose overly strict restrictions that blindly block all app background activities.

4.3 Understanding Diehard Behaviors

A comprehensive understanding of diehard behaviors can provide us insights on designing fine-grained app lifecycle control. We collect cases from popular user forums [6, 7, 4]

and well-known developer sites [47, 40]. We manually analyze 23 diehard apps reported by users and thoroughly inspect the techniques discussed among developers.

A key insight we learn from the analysis of existing techniques is that *diehard apps create high-priority app components and/or develop interdependence between component callbacks, between app components, or between other apps*.

4.3.1 Escalating Process Priority

To evade being killed, apps attempt to trick the system into believing they are important to serving the user. As a result, their process priorities will be escalated.

Foreground service. Normally, apps are put into the background when the user goes back to the home screen or switch to another app. Android, however, allows apps to start foreground services in which continuous tasks (*e.g.,* music playing, file downloading) will not be interrupted even the user is not currently using the app. Foreground services have much higher priority and therefore they will not be easily killed. This feature has been widely abused. Apps can simply call `startForeground(int, Notification)` to turn a background service into foreground state. The system considers foreground services to be user-aware and thus not candidates for killing even under heavy memory pressure. Before Android N there are bugs in displaying notifications, which are exploited to start foreground services stealthily without user awareness.

Floating view. Apps can keep a tiny, invisible floating view in the foreground, abusing the `SYSTEM_ALERT_WINDOW` permission. It is a known problem that this permission allows an app to draw overlays on top of other apps, and it is automatically granted for apps installed from Google Play [86].

Native process. Apps are allowed to run native executables using `java.lang.Runtime.exec()` APIs outside the app processes. Unlike app processes in the Android runtime, native processes are out of the control of the Android

framework's memory management. They by default have higher priority, especially when they run as daemons. The native processes may not be used to perform complicated tasks but rather to guard certain app components.

4.3.2 Auto-run

Apps utilize auto-run techniques in order to automatically start up after reboots and restart themselves after being killed. Different from escalating process priority, auto-run behaviors create interdependence between apps or between an app and the system. Even if the user uses task management tools [2, 19] to kill background apps, diehard apps can still manage to restart with auto-run.

Sticky service. A service makes itself "sticky" by returning `START_STICKY` from its `onStartCommand()` callback. A sticky service, if uses no other diehard techniques, will be recycled by the system when available memory is low. However, the system recreates the sticky service once it gets out of the low-memory state.

Listening to system events. The system sends out broadcasts when certain events occur. Apps that are interested in specific events get notified if they have registered corresponding broadcast receivers. For example, `SIG_STR` is broadcasted out when signal strength changes, and an app listening to this broadcast will be awakened. When a receiver is registered in the manifest and the app is not running, a new process will be created to handle the broadcast. This gives the app the chances to start other components thereafter.

Watchdog. A watchdog process is used to monitor the process that needs to keep alive. If the process being watched is dead, the watchdog restarts it immediately. Watchdog processes are usually implemented as native processes, and there are several ways to monitor another process' state (*i.e.,* running or dead). For example, the watchdog could be a native daemon which establishes a local socket channel with the app process [129]. If the socket

channel is somehow broken, it means the app process is dead. In this case, the native daemon tries to restart the app process immediately.

Abusing account synchronization. Apps are allowed to create a sync adapter component that encapsulates the code for the tasks that transfer data between the device and a server. Based on the scheduling and trigger provided, Android's sync framework runs the code in the sync adapter component (no matter the app is running or not), from which other components of the app can be started.

Scheduled tasks. AlarmManager allows scheduling an app to be run at some specific point in the future, even if the app is not currently running. JobScheduler first became available in Android 5.0. Apps register jobs, specify their requirements for network and timing. The system then schedules the jobs to execute at the appropriate times. Both AlarmManager and JobScheduler are abused by apps to realize auto-run. Observables can also be used to set up periodically tasks.

Cross-app wakeup. Apps developed by the same developer and apps integrating the same SDK can work together to keep long-running. For example, all Baidu apps have the same *ShareService* which periodically looks up other Baidu apps installed on the device and tries to bind to them. Since the system starts the target app for completing inter-app ICCs, one running Baidu app can thus *wake up* all other Baidu apps that are not running.

Explicitly invoking lifecycle callbacks. The system manages app component lifecycle. Different callbacks are invoked by the framework at each stage of an app component. For example, when the system destroys a service, `onDestroy` is called. The purpose is to give apps an opportunity to save running states and die gracefully. Apps are able to override these callbacks. They can thus abuse them by explicitly calling `onStart()` inside `onStop()` so that the component will not finish. The behavior shown in Figure 4.1 adds an edge to the lifecycle, creating a cycle.

4.4 Fine-Grained Lifecycle Control

Based on the insight we learn from diehard apps and their behaviors, we believe that in an appropriate graph representation, high-priority components can be identified as special nodes and the interdependence can be captured as cycles. We propose the Application Lifecycle Graph (ALG) to accurately describe apps' lifecycles as a whole in a fine-grained manner. Diehard behaviors are reflected on the ALG as either edges with particular properties, or cycles indicating the interdependencies between apps, app components, or component callbacks. The benefits of ALG are two-fold. First, it can capture and record all app and system events (*i.e.,* edges on the graph) that affect lifecycle. Second, it allows us to convert problems such as diehard behavior detection into graph-based problems, *i.e.,* cycle detection. We design a runtime framework that utilizes ALG to dynamically track app states and realize fine-grained lifecycle control, which overcomes limitations of static analysis based approaches that lack efficiency, scalability, and extensibility. The framework also exposes the ALG and lifecycle control capabilities as a set of APIs in order to facilitate the development of new functionalities.

4.4.1 Application Lifecycle Graph (ALG)

ALG models lifecycles of all installed apps in three layers. From the higher level to the lower level, they are (1) cross-app ICC graph, (2) intra-app ICC graphs, and (3) component callback graphs. We have

$$ALG = (\mathcal{N}_{app}, \mathcal{E}_{cross-app-icc})$$

where $\mathcal{N}_{app}$ is the node set and $\mathcal{E}_{cross-app-icc}$ is the edge set. Each node represents an installed app: $\mathcal{N}_{app} = \{G_{app_{0\ldots n}}\}$, $G_{app_i, i\in[0,n]}$ is the intra-app ICC graph of app i. Each edge represents a cross-app ICC event. We further define $G_{app_i} = (N_{comp}, \mathcal{E}_{intra-app-icc})$, and $\mathcal{N}_{comp}$ is a set of nodes representing app components: $\mathcal{N}_{comp} = \{G_{app_i comp_{0\ldots m}}\}$. The edge

set, $\mathcal{E}_{intra-app-icc}$, represents intra-app ICCs. $G_{app_i comp_j}$ ($i \in [0, n], j \in [0, m]$) is the callback graph of component j in app i. Nodes of a callback graph are callback methods, while edges are call sequences of those callback methods: $G_{app_i comp_j} = (\mathcal{N}_{callback}, \mathcal{E}_{method-call})$.

Figure 4.2 illustrates the ALG structure. The top level is a graph consisting of apps (also the Android framework, which will be discussed in §4.4.1.1) and cross-app ICCs. For example, app_0 starts app_1 with a cross-app ICC. Each app node is actually an intra-app ICC graph, whose nodes are either app components or native binaries, and edges are intra-app ICCs. For example, app_0 has three components, among which $component_0$ starts $component_1$ and $component_1$ starts $component_2$. Each component has a callback graph that models its callback sequence.

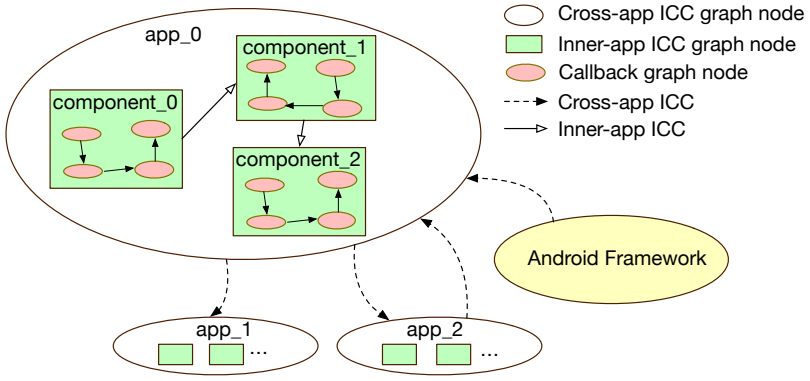

Figure 4.2: An ALG illustration. The Android framework is represented as a special node in the same level as apps. Edges have attributes that provide event contexts.

4.4.1.1 Abstract the Android Framework

The entire Android framework is abstracted into an app node in ALG, although the Android framework consists of a number of different packages and system services distributed into multiple processes. The rationale behind this abstraction is that from the apps' perspective, ICCs between the framework have no difference compared to cross-app ICCs.

Apps may interact with different system services during their lifetime. For instance, as described in §4.3, apps register their components to the framework, and the framework will start those registered components when required conditions are met. There are also many

built-in system apps that normal apps can communicate with. Aggregating framework packages into one single node reduces graph complexity by eliminating unnecessary nodes and edges, and significantly speeds up operations on the ALG. Meanwhile, system services and system apps are critical to normal functioning of the system; they are out of the scope of our fine-grained lifecycle control. This abstraction does not have a negative impact on the precision of the ALG.

4.4.1.2 Lifecycle Event Context

Edges in ALG represent lifecycle-related events. We provide event context as edge attributes. We consider the four categories of attributes: (1) User interaction, *i.e.,* whether or not an app or a component is initiated by the user is an important factor for determining the legitimacy; (2) Frequency, *i.e.,* the frequency of an ICC event indicates how aggressive an app is in terms of being diehard; (3) ICC type, including app status (*i.e.,* foreground or background), triggering method call (*e.g.,* `startActivity`, `bindService`); and (4) Status, *i.e.,* enabled or disabled, for enforcing lifecycle control policies. For example, in Figure 4.3, ICC edges provide information on how a component is started, and what shell command is executed to start a native component. Similarly, in Figure 4.4, cross-app ICC edges have information on the interactions between apps and the Android framework.

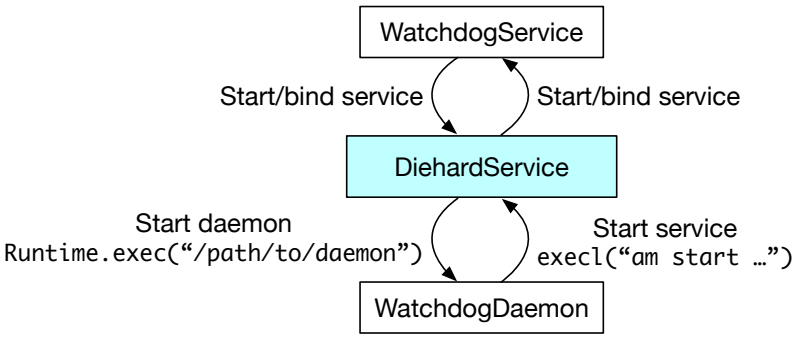

Figure 4.3: Partial ALG: intra-app ICC graph for an app having watchdog component. Irrelevant ALG parts are omitted.

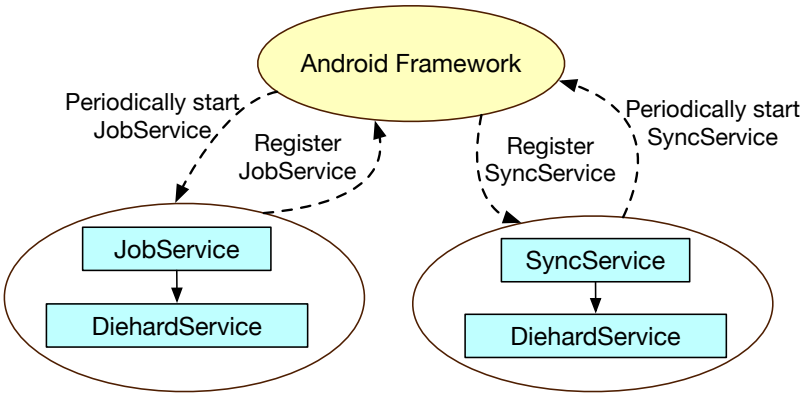

Figure 4.4: Partial ALG: cross-app ICC graph capturing scheduled task and account sync. Irrelevant ALG parts are omitted.

4.4.2 Fine-grained Lifecycle Control

To realize fine-grained, component-level app lifecycle control, we propose a lightweight runtime framework that builds ALG on-the-fly and exposes APIs to support the development of new functionalities. The advantage of a runtime system is that it can capture genuine runtime information, thus ensure accuracy, although performance overhead is inevitable and completeness cannot be guaranteed. Pure static app code analysis can gather relatively comprehensive app behaviors, but it is not precise due to the lack of source code and its inherent limitations that cause over-approximation, *e.g.,* points-to analysis [56]. The design of the framework must satisfy the following requirements.

1. Non-blocking monitoring. To reduce app perceived delay, we must not block app executions. This brings challenges in placing hooks in the Android framework for collecting runtime information.

2. ALG accuracy. Our lifecycle control framework relies heavily on the ALG. An accurate ALG is the foundation of new functionalities developed on it. However, there is no existing mechanism in the Android framework to support ICC caller component identification. In fact, very limited caller information is available, including only app UID, PID, and package name.

3. Nondisruptive control. If an app or an app component is being restricted, we need to gracefully shut it down, without causing crashes that will be perceived by the user.

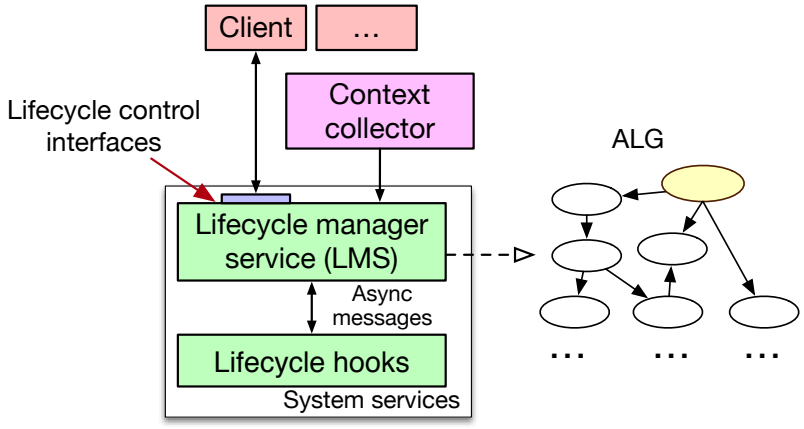

Figure 4.5: Overview of the framework. There could be multiple client apps that use the lifecycle control APIs.

Figure 4.5 depicts the architecture of our proposed framework. We add a system service, Lifecycle Manager Service (LMS), into the Android framework to maintain an ALG at runtime. To collect runtime lifecycle information we place various hooks into existing system services such as Activity Manager Service and Job Service. All hooks report collected data to LMS, which updates the ALG accordingly. Meanwhile, LMS exposes a set of APIs that provide the ALG and fine-grained lifecycle control capabilities to apps. We overcome the challenge of accurately identifying the caller component of an ICC using a Context Collector (§4.4.2.2). These interfaces enable various use cases. System-level developers (*e.g.,* device vendors) can leverage them to restrict diehard behaviors. Developers of task manager apps and battery saver tools can use the interfaces to better manage running tasks and to implement more effective battery saving policies.

4.4.2.1 Lifecycle Manager Service (LMS) and Hooks

LMS works with lifecycle hooks to build and maintain the ALG. Hooks are placed into several system services for collecting runtime app lifecycle information and controlling lifecycle events. We choose to hook services instead of app logic for two reasons. First,

Table 4.2: APIs provided by our framework for fine-grained app lifecycle control. `Bundle` objects are essentially key-value pairs. They are used to update one or multiple edge/node properties at a time.

APIs	Types	Description
`AppLifecycleGraph getLifecycleGraph()`	Sync	Return a copy of ALG
`AppCompGraph getAppCompGraph(String pkg)`	Sync	Return an app component graph with given package name
`CompCallbackGraph getCompCallbackGraph(String pkg, String comp)`	Sync	Return callback graph of an app component
`void setAppProperties(Bundle p)`	Async	Set properties of an app node on the ALG
`void setAppComponentProperties(Bundle p)`	Async	Set properties of an app component
`void setCrossAppEdgeProperties(Bundle p)`	Async	Set properties of an cross-app ICC edge
`void setIntraAppEdgeProperties(Bundle p)`	Async	Set properties of an intra-app ICC edge
`void setCompCallbackEdgeProperties(Bundle p)`	Async	Set properties of component callback graph edge

Android adopts a client-server model where apps send their requests to handling services. For example, ICCs are eventually executed by Activity Manager Service, which acts like a switch that looks up the target app and component, thus connects the caller and the target. Placing hooks in services allows us to centralize monitoring and enforcement logic so that we can keep minimal communication channels with LMS and therefore reduce overhead. Second, we cannot trust information coming directly from the apps, because apps have the capability of manipulating its own memory and bypassing the hooks.

For an operation that affects app lifecycle, there could be many intermediate procedures (*i.e.,* method calls) between the API being called and the internal method that eventually performs the intended operation. To balance overhead, accuracy, and extensibility, we must carefully choose appropriate locations for installing hooks. In general, we have three different hook placement options, as illustrated in Figure 4.6.

Close to the caller. If hooks are close to the caller, we can more easily collect the calling app UID, PID, and package name, as they are still available until the AMS calls `clearCallerIdentity`. Nonetheless, not all lifecycle-related operations can finally reach their targets, because they could fail at any intermediate method calls. As a result, we would collect false-positive ICCs and create ALG edges that do not really exist.

Close to the target. Lifecycle hooks can also be placed close to the target component. In this case, we would have very accurate information about the target without additional

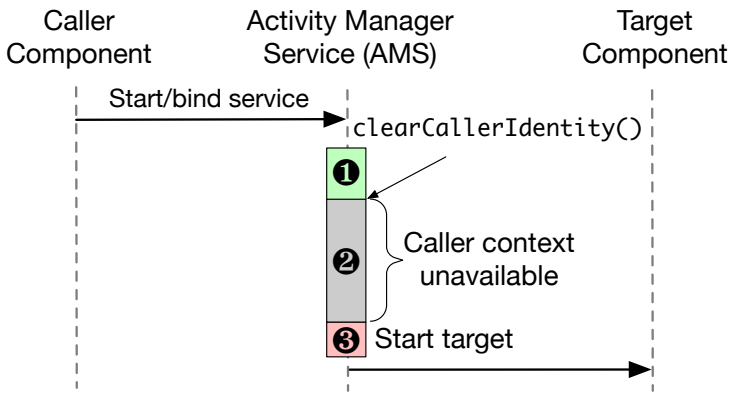

Figure 4.6: Hook placement options during service launching/binding ICC. The identity of the caller app is completely unavailable after the AMS calls `clearCallerIdentity()`.

efforts. The downside is we lose the caller app and component information completely.

Somewhere in-between. Placing hooks at certain points of the intermediate method calls allows us to balance accuracy and overhead. We can keep caller information before it gets cleared, and reuse intermediate return values to obtain target component identity. However, this option requires efforts in understanding system services code, which may change drastically across different Android versions. The cost of maintenance is the highest.

We choose to place lifecycle hooks close to the target, as our top goal is to ensure accuracy and eliminate false positives. The tradeoff is that we need to store caller information before it is cleared. Evaluations in §4.5 show that the overhead for this tradeoff is totally acceptable. Hooks could also be placed at both the caller side and the target side, but this would result in higher overhead.

4.4.2.2 Identify Caller Component

In Android's client-server model of app-framework interactions, apps are identified by their UID, PID, or package name, which means system services see all components of an app as a whole. The granularity of all access control mechanisms is per app, not per app component. As for our lifecycle control framework, we aim to achieve component-level

granularity, and the ALG requires caller component and target component for each ICC. Identifying the target component is trivial, but accurate identification of caller component is challenging.

To overcome this challenge, we modify base app component classes to attach caller component information automatically, as illustrated in Figure 4.7. Since everything from the app side could be manipulated by the app itself, we validate received caller component information in LMS. Specifically, all app components extend base classes from the Android SDK, *e.g.,* `android.app.Service`, `android.app.Activity`. We add into base component classes a `getIdentity()` method that returns class full name, including the package. Leveraging the polymorphism feature of the Java language, calling the method on concrete sub-class instances returns specific component names. The whole process is

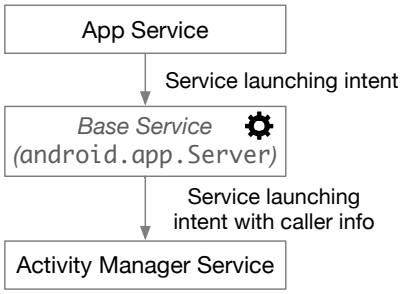

Figure 4.7: Attaching caller component information (using service as an example).

completely transparent to app and it requires no effort from app developers.

Everything coming from the app side cannot be trusted, because apps have the capability to manipulate anything in their own memory space. This means caller component information from the client side could be manipulated if the app wants to bypass or trick our caller identification method. To mitigate this potential problem, LMS validates caller component information it receives. First, the caller component must belong to the caller app, who can be identified by caller UID or package name. Second, the caller component must have been started already.

4.4.2.3 Nondisruptive Control

In addition to monitoring app lifecycle and building the ALG, our framework provides fine-grained, component-level lifecycle control. The idea is that lifecycle control policies can be stored as ALG node and edge properties. For example, if a service component is considered to be diehard we can simply set its `enabled` status to `false`, and let lifecycle hooks enforce it. We cannot return an error or throw out an exception within the hooks, because the hooks have no idea whether the caller app is able to properly handle the errors or exceptions. To avoid crashing the caller app unexpectedly, the hooks redirect ICCs to dummy components created by LMS. Those dummy components only execute minimal code and exit immediately.

4.4.2.4 Asynchronous Operations

To avoid hooks blocking the execution flow of apps, it is important to reduce the running time of hooks. Considering that hooks send ICC information to LMS and LMS takes time to process it, we let all hooks send asynchronous messages to LMS. The caller side (*i.e.,* hook points) does not have to wait for return values before proceeding. Moreover, whenever there is an update on ALG, LMS client should be aware of it. Instead of clients keeping polling ALG from LMS interfaces, LMS sends out a permission protected broadcast to notify clients. The protected broadcast can only be received by apps that have been granted the permission `android.permission.LIFECYCLE_UPDATES_ACCESS`, and user consent is required to grant an app this permission.

4.4.2.5 Exposed APIs

Table 4.2 lists a set of APIs that our framework provides to clients. To protect them from being abused, we enforce the permission `android.permission.LIFECYCLE_GRAPH_ACCESS`. This permission also requires user consent in order to be granted to an app. There are two categories of APIs according to how results get returned, *i.e.,* syn-

chronous and asynchronous APIs. The principle is that reading operations are synchronous and writing operations are asynchronous. In this way, the ALG obtained by clients are consistent with the one in LMS, and updating ALG does not block the caller components. The code snippet in Figure 4.8 shows how easy it is to use the APIs to query different levels of graphs from ALG and detect cycles.

```
void detectCycles() {
  // detect cycles on component callback graph
  for (app : installedApps) {
    for (comp : getAppComponents(app)) {
      callbackGraph =
          lms.getCompCallbackGraph(app, comp);
      bfs(callbackGraph);
    }
  }
  // detect cycles on inner-app ICC graph
  for (app : installedApps) {
    compGraph = lms.getAppCompGraphs(app);
    bfs(compGraph);
  }
  // detect cycles on cross-app ICC graph
  bfs(lms.getLifecycleGraph);
}
```

Figure 4.8: Querying different levels of lifecycle graphs and detecting cycles. Certain variable types are omitted. `lms` is a reference pointing to the Lifecycle Manager Service.

4.5 Evaluations

We implement the proposed framework on AOSP 8.0.0_r4 codebase and install it on a Nexus 6P Android phone with 3GB memory. In this section, we evaluate the accuracy of our approach to building ALG and the performance of the fine-grained lifecycle control framework. To demonstrate the usability and the capabilities of the APIs, we showcase two example client applications. We also present our findings based on the analysis of 17,598 applications from Google Play and a third-party application market.

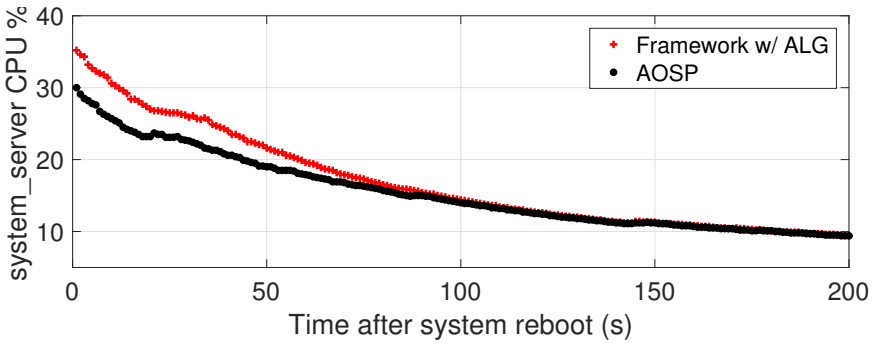

Figure 4.9: `system_server` CPU usage after device reboot.

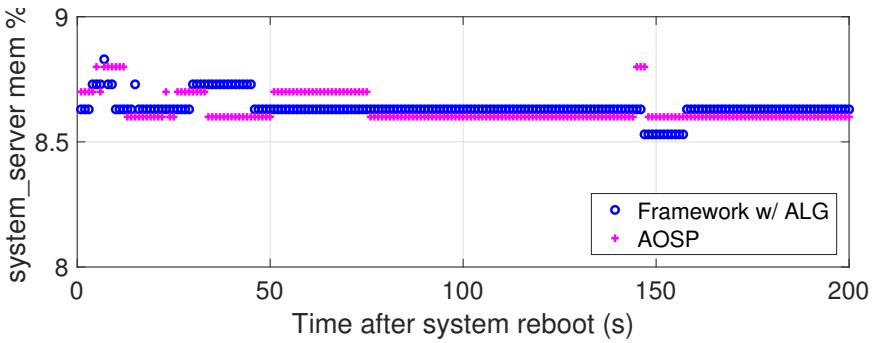

Figure 4.10: `system_server` memory usage after device reboot.

4.5.1 ALG Accuracy

We use the analysis results of the applications described in §4.2 as ground truth to evaluate ALG accuracy. Since the ALG is built from runtime information collected by the hooks, there are no false-positive ICCs. Our hook placement strategy ensures accurate identification of ICC target components. The only factor that could result in inaccuracy is our caller component identification approach. In our experiments, the framework captures 149 unique ICCs and accurately identifies all of their caller components.

4.5.2 Overhead

Lifecycle hooks and LMS are integrated into the Android framework, running in the `system_server` process. We compare CPU and memory usages with the original AOSP build, both have the same set of applications installed. At the very beginning of device booting, the lifecycle control framework adds approximately 5% CPU usage, as Figure 4.9

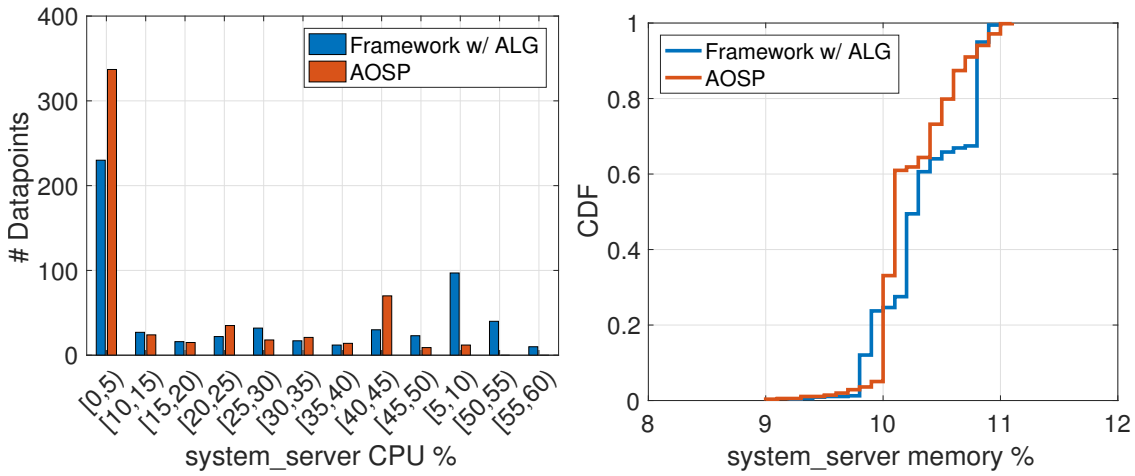

Figure 4.11: `system_server` CPU and memory usage while repeatedly launching applications.

depicts. This is reasonable as the initialization of the LMS takes additional CPU times, and the difference becomes negligible after around 100 seconds Figure 4.10 shows that the framework imposes only 0.15% additional memory usage. Most of the additional memory is used for storing the ALG at runtime, whose size is less than a few megabytes, depending on the number of installed applications. This is acceptable even on low-end devices with much less memory. We repeatedly launch an application 560 times to measure CPU and memory usages during application launches. Results are shown in Figure 4.11. The framework incurs less than 20% peak CPU usage, due to a high number of ALG updates. Still, the memory usage difference is small.

We also evaluate application launch time and system boot time with and without the proposed framework. We follow the official recommendation on launch time performance measurements [23]. Results are presented in Figure 4.12(a). The median application launch time of AOSP is 363ms, while our framework increases that by 93ms. This small change can be barely noticed by users. We then reboot the device 100 times to measure system boot time. Results shown in Figure 4.12(b) suggest that boot time increase is also insignificant. The median increases from 28.345s to 30.932s. As the number of installed applications increase, the time of reading ALG also increases, as depicted in Figure 4.13. This is be-

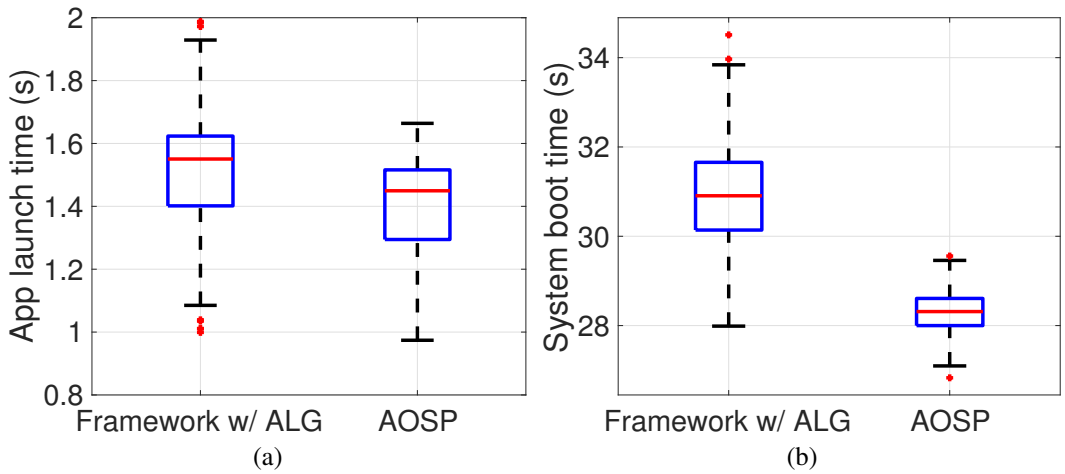

Figure 4.12: The comparison of application launch time and system boot between our framework and AOSP.

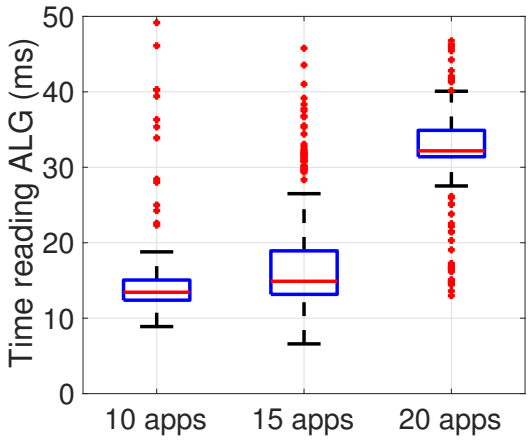

Figure 4.13: Difference in ALG reading time with different numbers of applications installed on the device.

cause the more applications are installed, the more complex the ALG is. Figure 4.14 is the cumulative distribution of time consumed at hooking points. 99% hooks are executed within 2 milliseconds.

4.5.3 API Usability

We implement two example client applications that leverage lifecycle control APIs to (1) detect and report cycles on ALG and thus detect diehard behavior and (2) restrict background ICCs that launch applications without user interactions.

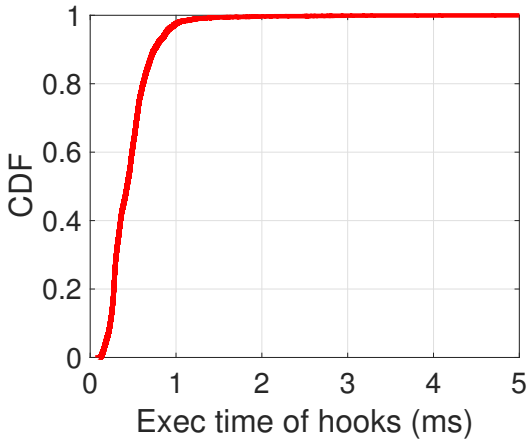

Figure 4.14: The cumulative distribution of hooks' execution time.

Leveraging event contexts provided by ALG as edge properties, we implement an application that demonstrates the effectiveness of fine-grained lifecycle control, also using the interfaces provided by our framework. We enforce a policy that prevents an invisible application component (no matter it is in the background or in the foreground) to launch inactive components in other applications. We fully charge the device, reboot it, and leave it for five hours without performing any operations on it. We measure battery level changes and battery discharge rate every 10 minutes with the Battery Historian tool [15]. The results presented in Figure 4.15 suggest that by the restriction of diehard behaviors is effective. Battery life can be extended significantly. Similar to CPU and memory usages after reboot, the discharge rate is higher with the framework at the very beginning due to additional initialization efforts.

4.5.4 Diehard Applications in the Wild

To the best of our knowledge, there is no prior study on diehard applications and their behaviors. To understand diehard behaviors in the wild, we analyze a large number of applications downloaded from Google Play and a third-party application market[1]. Due

[1]Google Play is inaccessible for Chinese users. `http://www.appchina.com/` is one of the most popular 3rd-party markets according to Alexa Rank.

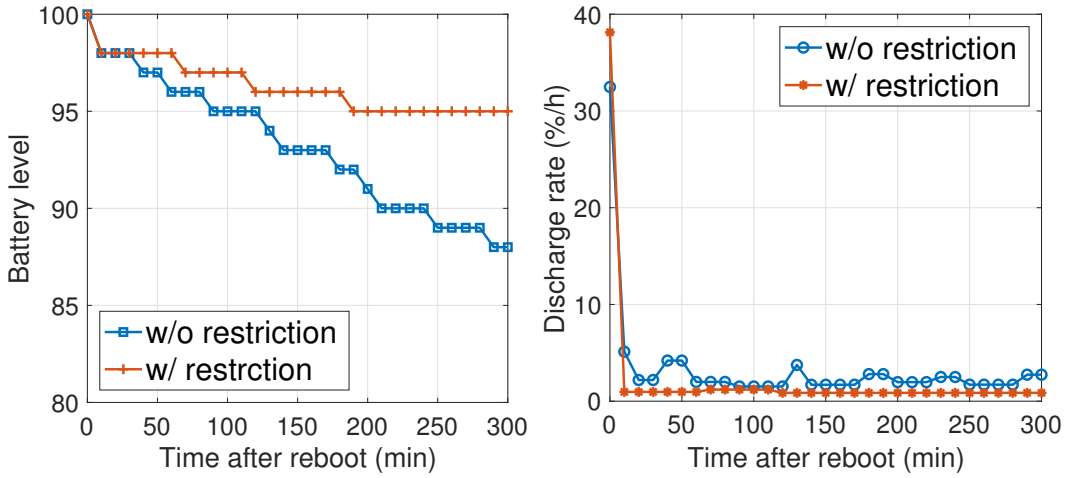

Figure 4.15: Battery life can be extended if diehard behaviors are restricted.

to the lack of an update-to-date Google Play dataset[2], we choose to download Google Play's top 500 best selling free applications from each of the 29 categories[3] by ourselves. 11,339 were successfully downloaded in early June of 2018. We also collect 6,259 best selling applications from the third-party application market covering all its 15 categories, excluding duplicated ones that also appear in Google Play. We first group applications by their developers and then install applications in the same group altogether. We use `aapt` to identify application user interfaces (*i.e.,* Activity components), and use the command line tool `am` to launch them, mimicking user interactions. Application analysis results show that diehard behaviors are common in both Google Play applications and applications from the third-party market. We also find diehard behaviors coming from widely used SDKs, although some of the host applications are not intentionally diehard.

Table 4.3 lists the percentages of applications that use each diehard technique. It is obvious that all diehard techniques except for account sync are more widely used by applications from the third-party market. Sticky service is the most prevalent among applications from both application markets. The percentages of Google Play applications having

[2]The popular PlayDrone application dataset [139] used in other work is very outdated (last updated in Nov. 2014).

[3]The category `ANDROID_WEAR` is excluded. Android wear applications are currently out of the scope of our study.

a floating view, native process, and explicit callbacks are significantly lower than that of applications from the third-party market. Figure 4.16 is the cumulative distribution of numbers of diehard techniques applications use. 38% Google Play applications have no diehard behaviors at all, while only 17% applications from the third-party market are non-diehard. These numbers clearly suggest that Google Play applications tend to be less aggressive in keeping themselves long-running.

Table 4.3: Percentage of applications that use each diehard technique.

Technique	**3rd-Party Market**	**Google Play**
Foreground service	16.3%	13.1%
Native process	5.6%	1.0%
Floating activity	25.2%	9.3%
Sticky service	29.3%	25.1%
System events	19.4%	18.5%
Watchdog	7.3%	2.8%
Account sync	0.3%	0.3%
Inter-app wakeup	20.1%	17.5%
Scheduled tasks	0.9%	0.7%
Explicit callbacks	5.4%	1.8%

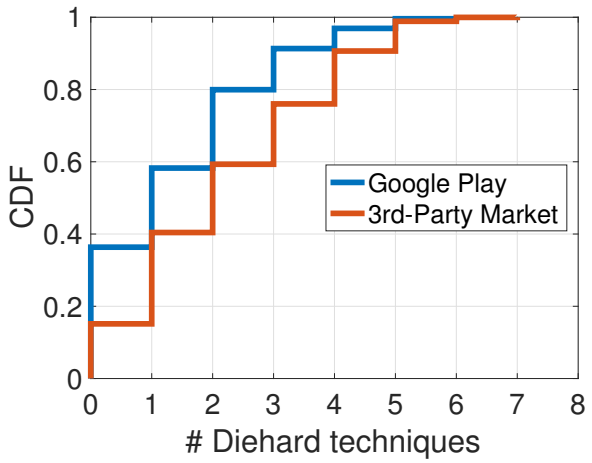

Figure 4.16: Numbers of diehard techniques used by applications from Google Play and the third-party market.

Table 4.4: Purposes of being diehard.

Purpose	# Applications	
	3rd-Party Market	Google Play
Sensor monitoring	40	61
Ads/promotions	59	51
Push notification	1,192	124
Keyguard	84	75
Hot patching	48	39
Downloading/uploading	280	57

4.5.4.1 Purposes of Being Diehard

We investigate the purposes of applications for being diehard. By manually examining the ALG and reversing APKs, we classify diehard behavior purposes into the six categories. Results are summarized in Table 4.4.

Sensor monitoring. Certain applications want to constantly monitor system events and user activities. Since Android N, most of the system broadcasts can only be received by dynamically registered broadcast receivers. For example, applications are no longer able to receive screen lock events with a static broadcast receiver. In order to monitor system events and take actions accordingly, applications have to keep alive so that their dynamic receivers are active. There are also applications constantly sensing the ambient environment, *e.g.,* lighting. Other examples include fitness applications that track user activities. They need to keep long-running in the background; otherwise, they would produce inaccurate results.

Displaying ads/promotions. A common business model for application developers is to make profits from ads displayed inside their applications. They usually keep an ad service running in the background to retrieve ads and show them to users. Some applications show promotion notifications from while running in the background. Such applications would like to be diehard so that they can maximize profits.

Push notifications. Google recommends Firebase Cloud Message (FCM) for sending push notifications to devices from the server side. However, FCM is not available in certain re-

gions (*e.g.,* China blocks Google services). The lack of OS-level message push service leaves developers no choice but to use third-party message push SDKs, or to implement their own message push services. One major metric for evaluating the quality of message push services is the delivery rate. To make sure push notifications can be delivered timely and successfully, push services have to leverage diehard techniques to keep themselves long-running in the background. We observe applications coming with multiple push services, all of which stay long-running in the background, creating several diehard services.

Keyguard. Non-system applications are not allowed to replace the lock screen. But applications can create UI components that look like a screen lock to users. In order to provide self-implemented keyguard functionalities while the device is locked, applications need a long-running service, which has to be diehard so that it can provide the required functionalities.

Hot patching. Android allows applications to load and execute dynamically at runtime. For example, they can download plugins in the format of `dex` files and load them by user demand. This feature is also used by some applications to realize hot patching. Users do not need to reinstall the application anymore. Instead, hot patching services can replace the out-dated code by updating the corresponding `dex` file.

Downloading/uploading. Certain applications download data from or upload local updates to their servers periodically. They use a diehard service to prevent the downloading/uploading process from being accidentally terminated. In fact, the recommended approach to downloading and uploading data is to use `AsyncTask`.

4.5.4.2 Third-Party Libraries

We find that applications may not intend to be diehard. Their diehard behaviors could come from third-party libraries. A summary of third-party libraries having diehard behaviors is listed in Table 4.5. To our surprise, we observe dedicated libraries that im-

plement state-of-the-art diehard techniques and their primary goal is to keep applications long-running. For example, a library with package name `com.daemon.keepalive` is found in several high-rating applications with millions of installs such as Smart Cooler [38], RAM Master [36], and SPARK [39]. An industry-leading Android anti-virus service provider, Qihoo 360, offers a malware scanning library with a diehard service `com.qihoo.magic.service.KeepLiveService`. This service appears in Qihoo family applications as well as non-Qihoo applications such as Super Antivirus Cleaner [41], which has a rating of 4.7 and more than 10 million installs. We also find an open-source daemon library that offers out-of-the-box diehard components [20].

Tencent Xinge is one of the most popular message push SDKs. It actively queries installed applications on the device and looks for applications that also have the same SDK integrated, *i.e.,* applications that also listen to the broadcast `com.tencent.android.tpush.action.SDK`. If another application is found to have Xinge SDK but is not currently alive, it tries to launch that application in various ways, one of which is presented in Figure 4.17. The built-in command line tool, `am`, is called to start the target Xinge services in other applications.

```
for (Map.Entry localEntry : localMap.entrySet()) {
  try
  {        Map key is app package name and value is service name.
    String str = "am startservice -n " + (String)localEntry.getKey()
    Process localProcess = Runtime.getRuntime().exec(str);
    int i = localProcess.waitFor();
    if (i != 0)
    {
      str = "am startservice --user 0 -n " + (String)localEntry.getK
      localProcess = Runtime.getRuntime().exec(str);
      i = localProcess.waitFor();
    }
```

Figure 4.17: Tencent message push SDK has diehard behavior that wakes up all its services using shell command `am` that can bypass background execution limitation.

Third-party libraries, in particular, those exhibiting diehard behaviors, are potentially leaking user privacy. SDKs such as Tencent Xinge, JPush, Xiaomi Push asks for sensitive permissions, including reading phone state, accessing WiFi/network state, accessing fine

location, and accessing coarse location.

Table 4.5: Third-party libraries coming with diehard behaviors, their purposes, techniques they use, and whether they request sensitive permissions.

SDK	**Purpose**	**Techniques**	**Sensitive Permissions**
Tencent Xinge	Message push, Notification	Native watchdog, foreground service, sticky service, system events, inter-app wakeup	Yes
JPush	Message push, Notification	Foreground service, floating activity, inter-app wakeup	Yes
iGenxin	Notification	Scheduled tasks, sticky service, foreground service, system events, watchdog	Yes
Baidu Share	Cross-app data sharing	Native watchdog, sticky service, foreground service, system events, inter-app wakeup	Yes
Xiaomi Push	Message push, Notification	Sticky service, foreground service, system events, inter-app wakeup	Yes
Eguan	Monitoring	Scheduled tasks, sticky service, foreground service, system events	No
EMChat	Notification	Scheduled tasks, sticky service, foreground service, system events, inter-app wakeup	Yes
Jiubang	Notification	Scheduled tasks, sticky service, foreground service, system events, watchdog	Yes

4.5.4.3 A Real-World ALG

Figure 4.18 presents a real-world ALG visualization from 10 applications. It shows the interactions between applications and application components, as well as lifecycle events. We find that Baidu applications perform cross-app wakeup intensively. Tencent applications tend to utilize framework services to be diehard. The ES File Explorer application (package name `com.estrong.android.app`) and Tencent Input application (package name `com.tencent.qqim`) have watchdog services.

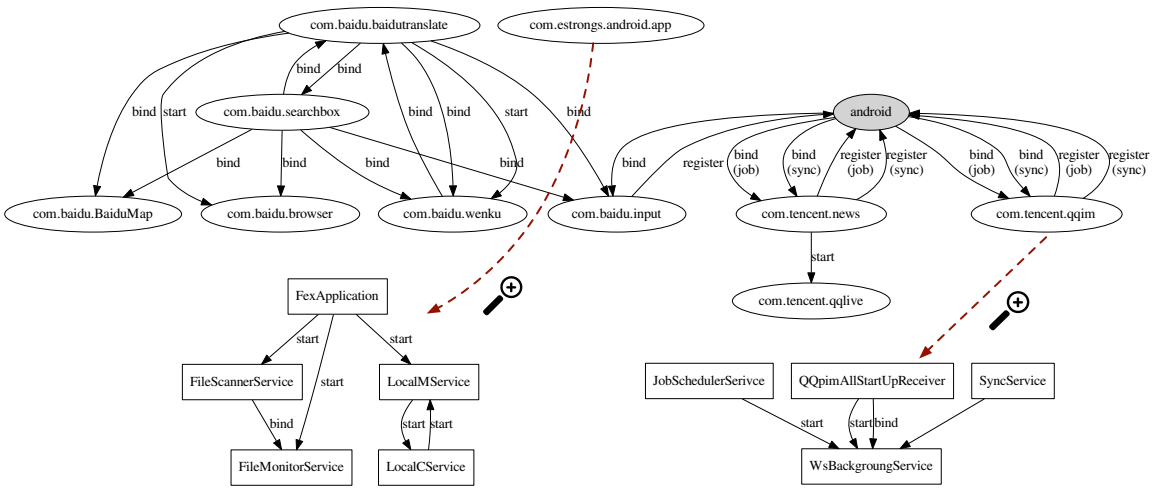

Figure 4.18: The topmost level (*i.e.,* cross-app ICC graph) of a real ALG visualized by Graphviz. `com.estrongs.android.app` and `com.tencent.qqim` are further inspected with one of their intra-app ICC graphs.

4.6 Discussion

While this work presents a fine-grained lifecycle control framework and makes the first step toward understanding diehard behaviors, there are limitations we plan to address in the future. First, we collected apps from Google Play and only one third-party market, which might lead to biases in results. We plan to do a larger scale study across multiple app markets. Second, due to the nature of runtime analysis, the framework cannot capture potential lifecycle events that are not triggered by the user, therefore the completeness of the ALG is not guaranteed. Third, a user study could be helpful for us to design better APIs for empowering app developers. As the wearable platforms become increasingly popular, we also plan to implement the framework on the Android Wear platform and investigate diehard behaviors of wear apps.

Legitimacy of diehard behaviors. We acknowledge that apps may have legitimate reasons for being diehard. For instance, a fitness app has to monitor user activities and locations constantly. However, we argue that apps should more gracefully achieve long-running and clearly indicate their background activities using Android recommended approaches, instead of abusing app lifecycle or gaming the system. Our proposed framework provides

foundations for developing robust diehard behavior detection and restriction mechanisms. Device vendors and developers can leverage the ALG and event contexts our framework provides to realize a crowd-sourced tool that can identify the legitimacy of diehard behaviors with a large dataset.

Background-running apps on iOS. Unlike Android, background processing in iOS is highly regulated. Long-running tasks require specific permissions to run in the background without being shut down, and only specific app types are allowed to do so [11]. Additionally, all iOS apps submitted to the App Store are manually reviewed to ensure that they do not violate Apple's guidelines. Developers are required to present a compelling reason for background activities. We argue that Android cannot simply adopt the iOS approach to restricting background executions. Android is meant to be a customizable platform and the whole ecosystem is open. Developers are believed to be responsible and follow the guidelines, which is unfortunately not true.

Circumventions. We place hooks into the system based on our understanding of current system implementation. In the future, new Android APIs might be introduced that could be abused to realize diehard behaviors and circumvent being captured by ALG. We argue that our framework is extensible and ALG can also be extended in order to adapt to future Android frameworks. As long as we build a runtime ALG, we can always rely on it and upgrade the detection algorithms.

Lessons learned. Benign diehard apps call for system-level support of long-running mechanisms that are transparent and controllable to users. Third-party libraries should be better inspected before being integrated, and library providers are supposed to provide configuration options to app developers. Nevertheless, it is challenging to prevent abuses of legitimate functionalities, because oftentimes this is a cat-and-mouse game and API designers are unaware of the potential side effects. Android API designers can leverage our framework to fix loopholes and better manage app lifecycle.

4.7 Conclusion

In this paper, we present a fine-grained app lifecycle control framework that leverages app lifecycle graph (ALG) to accurately describe app lifecycles. We overcome challenges such as caller component identification, nondisruptive app control. App study results suggest that diehard behaviors are common in apps from both Google Play and a third-party market. Evaluations show that our framework is capable of efficiently capturing app lifecycle events, imposing a negligible performance overhead. The proposed framework provides easy-to-use APIs on which new features can be developed. It empowers device vendors and app developers to leverage the ALG and event contexts to realize accurate detection of diehard behaviors and component-level, fine-grained control on app lifecycle.

CHAPTER V

Towards Secure Configurations for Real-World Programmable Logic Controller Programs

In this chapter, we present our study on the data access interfaces and their usage in modern Industrial Control Systems (ICS), where we identify insecure configurations in Programmable Logic Controller (PLC) programs that can be abused to cause safety violations. We manage to solve challenges in handling non-standard, vendor-specific program instructions, modeling controller scan cycle, and gathering safety-critical hardware output. This work can help ICS administrators and process engineers make secure configurations for PLC programs and prevent abuses.

5.1 Introduction

In recent years, industrial control systems are going beyond the factory floor. They incorporate industrial Internet-of-things (IIoT) devices to gather richer production data from plants, and utilize enterprise resource planning (ERP) tools to realize more intelligent decision making [147, 102]. These advancements would not be possible without the increasing *connectivity* and *interoperability* of the industrial control systems. Heterogeneous devices (*e.g.,* sensors, robots, motors, and drives) are able to communicate via industrial networks using dedicated protocols such as EtherNet/IP. Hardware tools, process controllers, and

software applications can interact using common languages. For example, modularized device functionalities are exposed so that they can be programmatically operated by process controllers. Meanwhile, the adoption of big data analytics requires more and more data to be collected from individual devices.

In support of the increasing connectivity and interoperability, not only between hardware devices but also across hardware and software boundaries, industrial equipment and software tools have to open more communication and control interfaces. As a matter of fact, industrial standards and specifications have been developed for use in field data retrieval and process control. The most popular ones are Open Platform Communications (OPC) [52] and MTConnect [51]. These technologies, together with modern PLCs, have brought flexibility, reconfigurability, and reliability to industrial control systems.

Unfortunately, those interfaces that provide the unified data access and control capabilities (1) inevitably expand the attack surface of industrial control systems and (2) drastically increase the complexity of controller programs. Many people still believe in the "air gap", a philosophy that says we can truly isolate critical systems from the outside world, but in reality air gaps do not work [65, 85]. This means the expanded attack surface in the industrial control systems, which are oftentimes isolated from the Internet, can still be abused and exploited by external attackers, not to mention insider adversaries. Compared to attacks targeting on software systems that often aim to make profits or steal data, cyber attacks on factory plants are intended to sabotage physical infrastructures. We have witnessed incidents, including Stuxnet [85], German Steel Mill cyber attack [105], and Ukrainian Power Grid attack [70], where external adversaries first managed to hack into the plants and then manipulated critical safety parameters, such as the frequency of nuclear centrifuges [104]. The increased complexity makes process control programming more error-prone. This can lead to anomalous automation behaviors that are difficult to debug and safety violations that sometimes cause fatal accidents [49, 50].

In view of these problems, we need a better understanding of today's PLC programs,

especially their potential security/safety loopholes resulting from insufficiently protected data access and control interfaces. Existing work on analyzing PLC programs goes with two major directions, both having significant limitations. First, there is existing research [98, 117, 118, 63, 62] that aims to verify PLC logic statically in a formal manner, but such static analysis techniques suffer from significant false positives due to the lack of runtime physical contexts. Because of the condition-based programming model of PLC programs, static analysis approaches may find potentially problematic code paths that are never executed at runtime. For example, a piece of code leading to a safety violation is triggered when the conveyor speed exceeds 20 miles per hour. However, due to the physical constraints of the motor or the output limit of the drive, the conveyor may never go that fast. Second, to overcome the aforementioned limitations, prior work [81, 89, 95, 100, 114, 122] has explored the application of simulations and symbolic execution to identify safety violations. In spite of being effective in finding bugs in the internal logic of independent PLC programs, these techniques do not take into consideration the interoperations between PLCs and controlled devices, as well as software programs. In addition, the unavailability of real-world PLC programs hinders the practicality of existing work.

In this work, we conduct the first analysis of real-world PLC programs to detect potential safety violations caused by abuses of data access and control interfaces. Unlike prior work that mostly uses benchmark programs, programs on educational platforms, or experimental, software controller programs, we collect over 800 real-world PLC programs from the industry-leading PLC vendor, Rockwell Automation. We design and implement *PLCAnalyzer*, which translates Rockwell PLC programs into equivalent C code and then performs static taint analysis. Specifically, it detects code paths that start from exposed interfaces and eventually reach safety-critical variables, such as motor speed and drive voltage. Results produced by PLCAnalyzer are further validated, using Rockwell PLC emulators. Our findings reveal several common problems in the PLC programming paradigm. First, exposed data access and control interfaces are insufficiently protected, due to both

developers' ignorance and inadequate security mechanism of PLCs. This allows adversarial to easily modify system parameters and cause safety issues. Second, add-on instructions (AOI) being widely used in the industry have direct access to hardware, but they do not validate input at all, allowing adversaries to trigger safety violations by simply controlling AOI inputs.

We make the following contributions in this work:

- We study over 800 real-world PLC programs from various industries and identify common problems that have critical safety implications. To the best of our knowledge, this is the first work that uses such a practical dataset and reveals previously overlooked, non-contrieved issues.
- We present PLCAnalyzer, a tool that automatically analyzes PLC programs for safety-critical code paths that are reachable from exposed data access interfaces. We identify 176 critical paths among 433 complete programs.
- We identify AOIs that have direct access to hardware but perform no input validation. We propose and discuss defense mechanisms for preventing exposed interfaces from being abused and exploited.

5.2 PLC and Data Access

5.2.1 Programmable Logic Controller (PLC)

The primary use of PLCs is to control a collection of processes on the factory floors and plants. PLCs have been widely adopted as highly reliable automation controllers for a wide range of industries [53], such as chemical, automotive, oil and gas, energy, and so on. PLCs run the programs in an infinite loop where each iteration, namely a *scan cycle*, consists of three main phases.

1. Input. The PLC scans inputs from external sources (*e.g.,* RFID sensors, motors) and buffers them in its memory as a list of variables (also called tags).

2. Execution. The PLC runs programs from beginning to the end, calculating new variable values based on program logic. Variable values in the memory are updated.

3. Output. The PLC writes updated variables altogether to external sources. Then the PLC is ready for the next cycle.

IEC 61131-3 is the *de facto* standard that specifies the syntax and semantics of a unified suite of programming languages for PLCs. It defines five different programming languages, including three graphical languages (*i.e.,* ladder diagram, function block diagram, and sequential function chart) and two textual programming languages (*i.e.,* instruction list and structured text). However, the IEC 61131-3 specification only provides a minimum set of functionality that can be extended to meet end-user application needs. The downside is that each PLC vendor may implement different components of the specification or provide different extensions. Moreover, the instruction set specified by IEC 61131-3 is entirely optional [58]. All major PLC vendors have developed their own instruction sets. As a result, existing work [126, 125, 114] on static code analysis of IEC 61131-3 programs are therefore incapable of dealing with real-world PLC programs.

5.2.2 Data Access

Modern PLCs adopt tag name based memory mapping. A tag is the name for a specific memory location, and it also has a data type assigned. Process engineers can use tag names in their PLC programs to store and access data. In addition, tag names can also be bond to PLC inputs and outputs, making it easier to programmatically operate external hardware. There are two types of tags, *controller tags* and *program tags*. Controller tags are global variables that are visible for all devices on the same network via OPC, whereas program tags have a very limited scope that is only visible inside a PLC program. Both controller

tags and program tags have the an attribute that speicifies their accessibility, read-only (*i.e.,* `ro`) or readable/writable (*i.e.,* `r/w`).

OPC is a standard interface to communicate between numerous data sources, including devices on a factory floor, laboratory equipment, test system fixtures, and databases. To alleviate duplication efforts in developing device-specific protocols, eliminate inconsistencies between devices, provide support for hardware feature changes, and avoid access conflicts in industrial control systems, the OPC Foundation defined a set of standard interfaces that allow any client to access any OPC-compatible device through an OPC server. Most suppliers of industrial data acquisition and control devices, including PLCs, are designed to work with the OPC standard. The OPC Unified Architecture (UA), released in 2008, is a platform-independent service-oriented architecture that integrates all the functionality of the individual OPC classic specifications into one extensible framework. OPC UA starts providing limited security guarantees to authenticate client identify and authorize access requests.

5.3 Threat Model

Figure 5.1: An Allen-Bradley ControlLogix 5563 PLC.

We assume that the attacker has managed to hack into an industrial control system and

has compromised at least one device in the control network, but he/she cannot reprogram PLCs directly to inject malicious code that deliberately causes safety violations and physical damages. Instead, the attacker can abuse open but insufficiently protected data access and control interfaces to alter certain PLC program variables. This threat model is more realistic than existing work (*e.g.,* TSV [114], SABOT [113] and the reasons are two-fold. First, one real-world plants, new generation PLCs have a physical switch that prevents reprogramming during operation. As Figure 5.1 shows, the switch controls three modes, *RUN*, *REM*, and *PROG*. If the PLC is in the RUN mode, no changes can be made to the programs. It is impossible to reprogram the PLC if the attacker has no physical access. Second, PLC programs are relatively stable, and they do not need to be updated for months or even years. Therefore, the mode will not be updated frequently, leaving attackers tiny window for reprogramming the PLC.

5.3.1 Motivating Example

Exposed data access and control interfaces can be exploited to launch attacks. A concrete example from a genuine PLC program is illustrated in Figure 5.2. The controller tag `CNC1Bools.0` is set as externally readable and writable, and its value controls another tag, `RunPartOnCNC1`. When `RunPartOnCNC1` is `true`, `ToCNC1.Bools.0` becomes `true` as well. There exists a data flow `CNC1Bools.0` $\rightarrow$ `RunPartOnCNC1` $\rightarrow$ `ToCNC1.Bools.0`. Since `CNC1Bools.0` is exposed and could be controlled by adversaries in our threat model, the output `ToCNC1.Bools.0` therefore could be indirectly controlled by adversaries too. However, `ToCNC1.Bools.0` is safety critical because it controls the robot arm's behaviors. If it is `true` the robot picks up incoming parts from conveyor and drops them into the CNC. While the CNC is processing the current part, `ToCNC1.Bools.0` is supposed to be `false`, meaning that the CNC is not ready for handling the next part. An attacker, by controlling `CNC1Bools.0` exposed through the data access interface, can trigger an overloading attack on the CNC by deliberately chang-

ing its value to `true`.

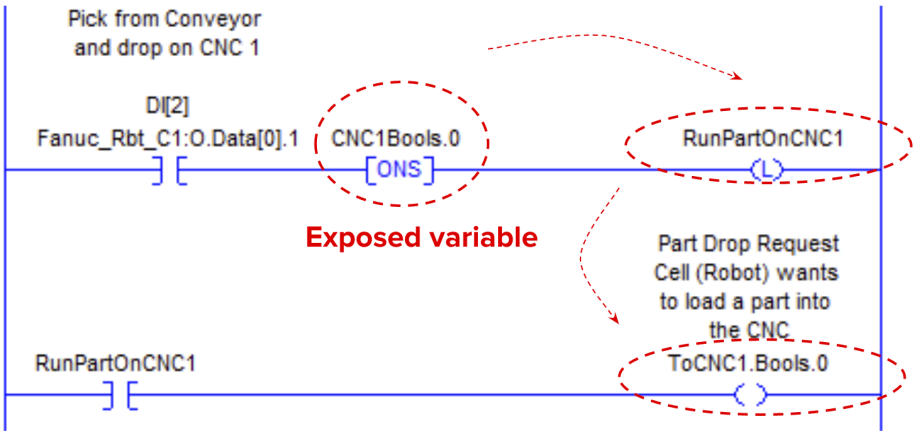

Figure 5.2: Example ladder logic with an exposed variable `CNC1Bools.0`.

5.4 Design and Implementation

Data access and control interfaces exposed on the control network but insufficiently protected can be abused to (1) access confidential production data and (2) trigger control program logic that can modify safety-critical system parameters. Compared to attacks that directly reprogram control logic, causing safety violations through the manipulation of tag values is much more subtle and stealthy, because there will not be visible changes happening.

As we have described in §5.2.2, a PLC program has controller tags that are globally visible on the control network. In this PLC programming model, there are two potential issues:

- If a tag is only used inside a program—meaning it is supposed to be a program tag that has limited visibility—but it is defined as a controller tag, internal program state or production data can be leaked.

- If a controller tag's access attribute is `r/w`, it is vulnerable to attacker manipulation. Once an attacker-controlled value propagates to a safety-critical physical out-

put, safety violation can be caused.

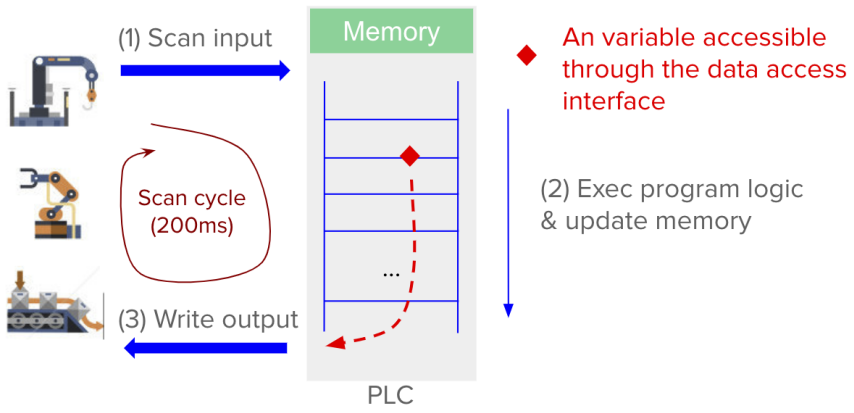

Figure 5.3: Safety-critical path illustration.

More specifically, we consider a tag to be *leaked* if its value comes from production equipment and it should be a program tag but in reality it is defined as a controller tag. We consider a controller tag to be *dangerous* if it is readable and writable, and its value eventually has an influence on any physical output. As illustrated in Figure 5.3, a *critical path* is a code path along which an exposed dangerous tag finally reaches a physical output.

To automatically and accurately identify leaked tags and dangerous tags, as well as critical paths in PLC programs, we propose PLCAnalyzer, a static analysis tool that translates PLC code into C equivalent and performs data flow analysis.

5.4.1 PLCAnalyzer Overview

The overall steps of PLCAnalyzer are shown in Figure 5.4. The PLC-C translator takes PLC code as input and generates the corresponding C equivalent. PLC data structures are transformed into C `structs`. Controller tags and program tags are converted into C global variables and local variables, respectively. Their access properties are recorded as well. Intra-procedural data flow analysis and inter-procedural data flow analysis as a whole identify all *leaked* tags, *dangerous* tags, and critical paths. An analysis report is generated at the end, which will be further examined manually.

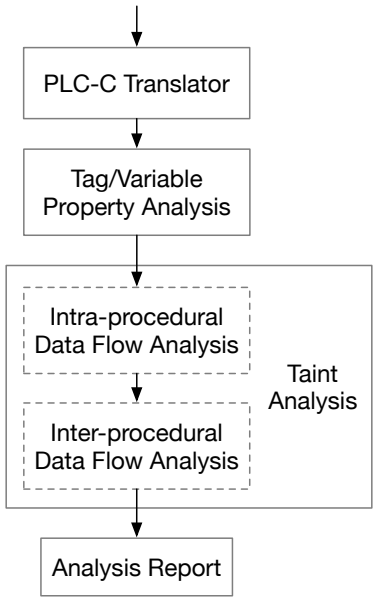

Figure 5.4: PLCAnalyzer analysis steps.

Identifying leaked variables relies on the knowledge of sensitive program inputs from controlled devices. For example, a quality control camera measures final product dimensions. This information is considered as a factory secret and therefore it needs to be protected. However, if a PLC program requires dimension data from quality control camera and the data is used somewhere else, this data could eventually flow to a global tag/variable that is globally visible on the control network. To detect such code paths, we need a list of sensitive inputs from controlled devices. Similarly, for detecting dangerous variables, we need a list of outputs that are finally written to physical outputs of controlled devices and change their physical states, such as motor speed. We obtain senstive inputs and outputs from hardware manuals.

5.4.2 Translating PLC Code

Our study emphasis is real-world PLC code that is beyond the IEC 61131-3 standard. Existing tools (*e.g.,* the Matiec compiler [24]) that are built to translate and compile PLC code complying the standard are therefore incompetent. We have implemented our own PLC-C translator, whose workflow is illustrated in Figure 5.5. Note that the Rockwell

PLC code is in `l5x`[1], a proprietary but well-documented format. Each program consists of multiple routines, which are similar to functions in C. The main routine is the entrance of a program, similar to the main function of a C program. After extracting tags and routines, we leverage *Instaparse* to develop a parser by describing the grammar in Backus–Naur form, as Figure 5.6 shows. The parser produces an abstract syntax tree (AST), on which we perform a semantic analysis to translate PLC instructions into C code. We implement the PLC-C translator in Clojure with 1,374 lines of code.

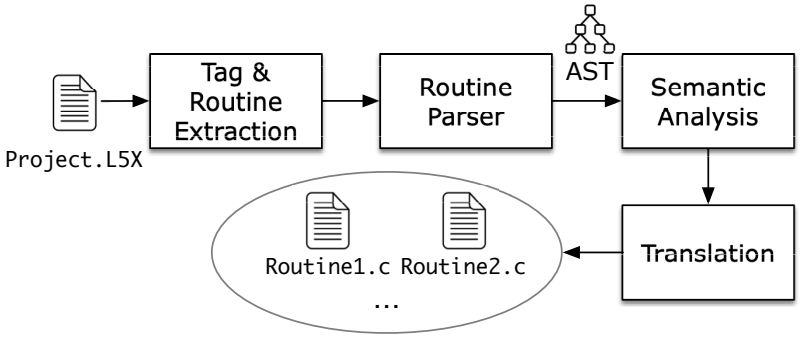

Figure 5.5: Translating PLC code into C.

```
<S> = routine
routine = rung+
rung = (instruction | branch)+ ';' <newline>*
branch-level = ((instruction | branch) ' '*)+
branch = '[' (branch-level ',')+ branch-level ']'
instruction-name = "EQU" | "XIC" | "CMP" | "XIO" | \
  "MOV" | "XIC" | "OTL" | "OTE" | "FLL" | "NEQ" | \
  "XIO" | "RES" | "LES" | "TON" | "CTU" | "NOP" | \
  "SUB" | "MUL" | "ADD" | "DIV" | "BTD" | "COP" | \
  "GSV" | "ONS" | "OTE" | "MOD" | "GEQ" | "CLR" | \
  "OTU" | "JSR" | "OSR" | "GRT" | "AFI" | "LEQ" | \
  "CPT" | "FOR" | "MSG" | "LBL" | "JMP" | "MID" | \
  "SWPB" | "DELETE" | "OR" | "AND" | "SBR" | \
  "RET" | "CONCAT"
instruction-arg = (tag-name | integer | float | '?')
instruction-args = instruction-arg (',' instruction-arg)*
instruction = instruction-name '(' instruction-args? ')'
tag-name = #'[a-zA-Z_]+[0-9a-zA-Z_\[\]\.:]*'
integer = '-' ? #'[0-9]+'
float = #'[-+]?[0-9]*\.[0-9]+'
newline = <'\r' ? '\n'>
```

Figure 5.6: Description of l5x grammar in Backus–Naur form.

[1] `l5x` is short for RSLogix 5000

5.4.3 Tag Property Analysis

Tags are defined in a dedicated section of `l5x` files. Besides name, each tag has several other attributes, including `Class`, `TagType`, `DataType`, and `ExternalAccess`. A controller tag is considered to be exposed if its `ExternalAccess` attribute value is "Read/Write." To further unveil the potential attack surface, we analyze the scope of tags. If a controller tag is only used by one routine, then it should be set as a local variable in this particular routine. If a writable global tag is ever written by only one routine, it does not have to support external access, meaning the `ExternalAccess` attribute can just be "ReadOnly." The analysis result generated in this phase will be used in the following steps for identifying critical paths.

5.4.4 Taint Analysis

There are two types of critical paths in the data flow analysis phase, *i.e.,* (1) intra-procedure critical paths and (2) inter-procedure critical paths. In an intra-procedure critical path, the starting point (use of global variables) and the ending point (the definition of a variable) are in the same procedure, and no call instructions are in this path. To analyze intra-procedure critical paths, PLCAnalyzer conducts data flow analysis to find all the definition dependence's reaching definitions, and repeat this process recursively, until it finds a use of global variables. In an inter-procedure critical path, the starting point (use of global variables) and the ending point (the definition of a variable) are in different procedures, and there are call instructions connecting the path. To find an inter-procedure critical path, PLCAnalyzer uses the information retrieved from intra-procedure critical path phase, also identifying intermediate paths from function arguments to variables definitions inside functions, and then iterates through the function calling tree to find out whether the intermediate paths can be connected and related to a global variable.

While the data flow analysis can reveal all the possible critical paths, there could be exposed variables that indirectly influence hardware output through control flows. For

example, if a condition of a branch instruction is on a critical path, and the condition being checked involves an exposed variable, an attacker can control which branch the program will go into at runtime. To capture exposed variables used in conditions on critical paths, PLCAnalyzer applies control flow analysis to find out the conditions of entering basic blocks, and labels those variable definitions in basic blocks whose conditions are on critical paths. Each labeled definition is also dangerous. Then PLCAnalyzer will update the set of dangerous definitions.

As PLCAnalyzer is adding new dangerous definitions during control flow analysis, it can possibly reveal more data flow critical paths. The analyzer will repeat data flow analysis and control flow analysis until the size of critical paths and the size of dangerous definitions do not grow anymore. We utilize LLVM to implement the static taint analysis on the translated C code, with around 1,000 lines of code.

5.5 Evaluation

5.5.1 Dataset

Due to the lack of a real-world PLC program dataset, prior work had to use contrived programs to conduct evaluations. We have managed to collect the first real-world PLC program dataset from Rockwell code repository, including over 863 user-submitted code samples. Figure 5.7 summarizes the categories this dataset covers, including programmable controllers (*i.e.,* PLCs), operator interface, partner products, software, motion control, and so on. The number of PLC programs is 458.

5.5.2 Results

Our results show that it is a common problem that PLC programs' tags have incorrectly configured visibility and accessibility. Among 458 PLC programs, 433 are complete and can be successfully translated into C. We identify 176 critical paths and 1,442 exposed

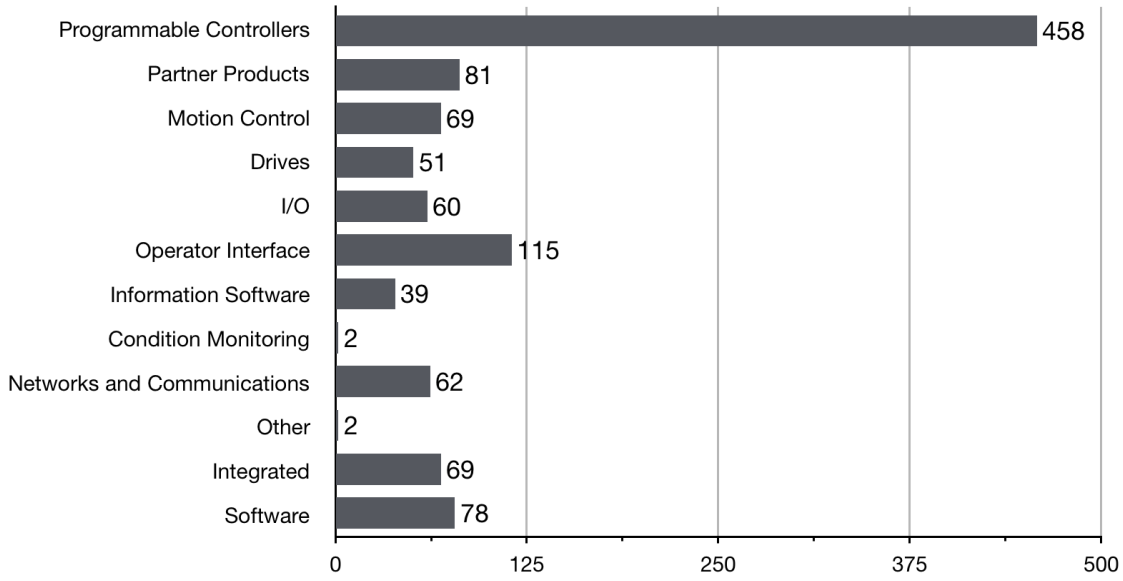

Figure 5.7: Dataset overview

PLC tags. The numbers of critical paths, grouped by PLC firmware versions, are shown in Figure 5.8. Add-on instructions (AOIs) are similar to software libraries that can be reused by different programs. We find 117 out of 388 AOIs directly control hardware outputs, but no safety violation checks are employed.

We take the program for the ArmorStart Motor Controller as an example to illustrate the functions of PLCAnalyzer. After converting the PLC code to C code (see Figure 5.9 and Figure 5.10), we apply PLCAnalyzer and obtain the analysis report. In the C code, there are several global variables at the beginning, translated from PLC tags. There is also a function `AS_284E_AOI` translated from the corresponding AOI. In the main function, global variables are initialized with default values and the `AS_284E_AOI` function is called in the end. Note that `scaled_speed`, an exposed global variable is passed to the function as the second argument. Inside the `AS_284E_AOI` function, the second argument `Inp_Scaled_Speed_At_400Hz`, is used to compute the physical output, `out_speed`. Since `Inp_Scaled_Speed_At_400Hz` is actually coming from the main function with the value of `scaled_speed`, an attacker can manipulate motor output by changing the value of the exposed variable `scaled_speed`. In fact, the value of `out_speed` depends

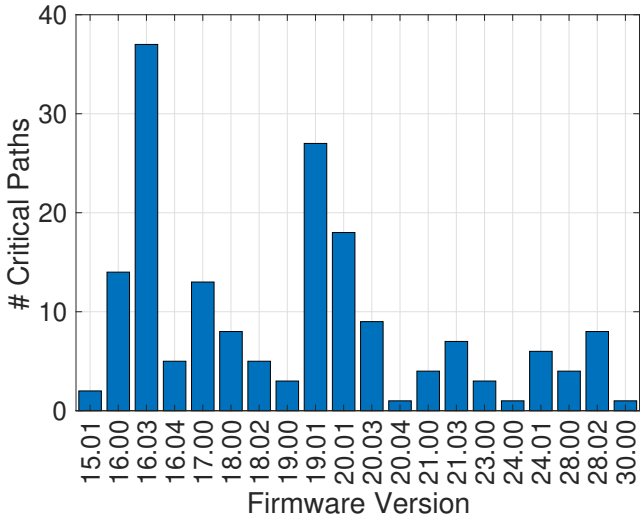

Figure 5.8: By PLC firmware version

on two variables (line 7 and 10 in Figure 5.10), but `Set_SpeedOper` is a local variable, *i.e.,* an unexposed tag, and attackers have no way to modify it. Based on the report, we can figure out a critical path, *i.e.,* `scaled_speed` → `Inp_Scaled_Speed_At_400Hz` → `out_speed`. Therefore, a potential safety violation is detected.

We model the PLC scan cycle as an iteration of an infinite loop. We define taint sources as globally visible, writable input variables exposed to attackers, and taint sinks as statements and instructions that modify the physical output. PLC programs have no pointer ambiguity, therefore point-to analysis is not needed.

5.5.3 Add-On Instructions (AOI)

In the dataset, we observe a number of AOIs, which are similar to libraries in software engineering. AOIs can be easily reused by other parties. Similar to importing a software library, PLC program developers can simply import compatible AOIs into their programs. Nevertheless, a potential problem of using AOIs is that AOIs usually take a reference to the hardware output module and update the hardware output inside AOI logic, which is a blackbox to PLC program developers. AOI implementations are supposed to be general, because AOI developers have no idea of the use scenarios. As a result, it is inappropriate

```
int scaled_speed;
bool Sts_OperatorModeEnabled;
bool Sts_ProgramModeEnabled;

int main() {
  Inp_AS_284E in;
  Out_AS_284E out;
  Sts_OperatorModeEnabled = true;
  Sts_ProgramModeEnabled = false;
  scaled_speed = 200;

  AS_284E_AOI(&in, scaled_speed); //Speed control
}
```

Figure 5.9: C code converted from a real-world PLC program: global variables and the main function.

to perform safety checks inside AOIs. For instance, depending on whether a motor is used to drive a drill or a conveyor, the speed ranges that are considered to be safe are different. Even for conveyors, in different factories or on different assembly lines, the safe speeds differ, too.

Besides `AS_284E_AOI`, we have identified six other AOIs whose outputs can be affected by attacker controlled tag values. They are summarized in Table 5.1. We find that the PowerFlex525 Variable Frequency Drive (VFD) has a variable `Freq` that controls its speed. Three other variables that are used to calculate `Freq` are `SpdRef`, `SpdScaler`, and `GearRatio`. An attacker can control drive speed by altering either of them, as these three variables are globally visible and writable to all devices in the same network. VFD speed is supposed to be within a safe range, and there would be a safety violation if it is too high.

5.5.4 Validating Results

Compared to software programs, PLC programs control heterogeneous hardware tools and the output is therefore highly dependent on the physical settings of a plant. Even though we have real-world PLC programs, we still cannot run them because of the lack of

```
void AS_284E_AOI(Inp_AS_284E *in, int Inp_Scaled_Speed_At_400Hz) {
  int Set_SpeedOper = 10;
  int Set_SpeedProg = 20;
  int out_speed = 0; // Output motor speed

  if (Sts_OperatorModeEnabled) {
    out_speed = Set_SpeedOper * 4000 / Inp_Scaled_Speed_At_400Hz;
  }
  if (Sts_ProgramModeEnabled) {
    out_speed = Set_SpeedProg * 4000 / Inp_Scaled_Speed_At_400Hz;
  }
}
```

Figure 5.10: C code converted from a real-world PLC program: the `AS_284E_AOI` function.

Table 5.1: AOIs that can be exploited by attackers to manipulate physical output.

Target hardware	**Purpose**	**Outcome**
45LMS Laser Measurement Sensor	Laser measurement sensor configuration	Incorrect measured distance. Changing imperial unit to metric unit.
PowerFlex525 Variable Frequency Drive (VFD)	VFD configuration	Taking over speed control.
AMCI 7662 I/O Module	Analogue data conversion	Modifying output data.
Mettle Toledo Scale	Scale setup	Causing inaccurate calibration.
MVI56-MCM	Modbus master/slave setup	Denial-of-service of slave devices.
1769-SM2 Multi-Drive Control	Connection configuration for AC drives	Controlling drive output.

controlled hardware. More importantly, it is impossible for us to set up an environment to run the programs, as we have no knowledge on what the real settings look like. PLC program developers only submitted their programs to the repository.

To overcome this challenge and validate the critical paths PLCAnalyzer has detected, we use Studio 5000 Logix Emulate, a commercial software product from Rockwell. From a PLC program's perspective, hardware input and output are also in the form of tags. Despite that we do not have required hardware, we can create tags in the same structures of hardware input and output, and these synthetic tags can be used to run the program. However, this emulator has strict firmware version control. The only PLC versions we can emulate are 28.00 and 28.02. We have identified four critical paths for version 28.00 and eight for

version 28.02: all 12 of them have been validated with the emulator.

5.6 Discussion

Limitation in the Common Industrial Protocol (CIP). Because of the limited capabilities of PLCs, existing industrial network protocols only support coarse-grained access control of tags. Figure 5.11 illustrates how data items are organized in CIP. On the CIP network, each device has its unique device ID. Data types are organized into classes (*i.e.,* Variable Frequency Drive) that have various attributes. There could exist multiple instances (*i.e.,* VFD1, VFD2) of a class. Current, CIP access control is per instance, meaning that even if only one attribute needs to be exposed, the entire instance has to be exposed. In other words, CIP does not support access control per object attribute, not to mention contextual integrity.

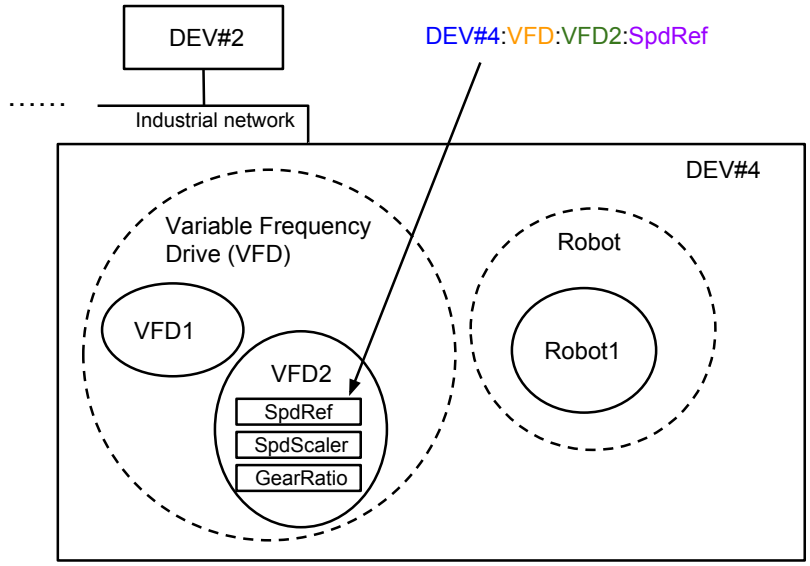

Figure 5.11: Data organization in CIP.

Fine-grained access control. The CIP limitation calls for a context-aware, fine-grained control framework, which is transparent to both factory floor equipment and controller programs. As real-world PLC firmware is proprietary, we are unable to make any changes

to it. Existing host- and network-based access control mechanisms do not suffice for ICS because they oftentimes overlook the underlying physical components. CPAC [84] is a cyber-physical access-control for energy management systems, but it requires firmware customization of controllers and is thus not practical. In contrast, we can enforce access control rules at the network layer and leverages policies that can be physical-device specific. We need to address challenges in efficiently intercepting and inspecting industrial network traffic, analyzing industrial protocols, and accurately identifying objects, instances, and attributes from packets. User-defined policies are applied at a per-packet basis at a vantage point where traffic from all the devices goes through, *e.g.,* the main switch. Policies should specify which device has access to which set of program variables. For example, only the human-machine interface can change variable `PowerFlex525`'s attribute `GearRatio` that can change drive speed in manual operating mode.

5.7 Conclusion

In this work, we make the very first step towards detecting potential safety violations caused by incorrect variable visibility and accessibility settings in real-world PLC programs. We design and develop PLCAnalyzer which employs control and data flow analyses to find out critical paths from globally visible and accessible PLC tags to safety critical physical outputs. However, our current prototype has limitations in finding device-specific critical paths related to physical properties and functionality.

CHAPTER VI

The Misuse of Android Unix Domain Sockets and Security Implications

This chapter presents our work on exposed native IPC endpoints in Android. We find both implementation and configuration issues in using Unix domain sockets among system daemons and applications. This work is the first systematic study in understanding the security properties of the usage of Unix domain sockets for cross-layer communications between the Java and native layers. We propose a tool called SInspector to detect potential security vulnerabilities in using Unix domain sockets through the process of identifying socket addresses, detecting authentication checks, and performing data flow analysis. Our in-depth analysis reveals some serious vulnerabilities in popular applications and system daemons, such as root privilege escalation and arbitrary file access. Based on our findings, we propose countermeasures and improved practices for utilizing Unix domain sockets on Android, from both implementation and SEAndroid configuration perspectives. This work can help application and system developers secure exposed Unix domain socket interfaces.

6.1 Introduction

Inter-process communication (IPC) is one of the most fundamental features provided by modern operating systems. IPC makes it possible for different processes to cooperate,

enriching the functionalities an operating system can offer to end users. In the context of Android, one of the most popular mobile operating systems to date, to support communications between different apps and interactions between different components of the same app, it provides a set of easy-to-use, Android-specific IPC mechanisms, primarily including Intents, Binder, and Messenger [9, 29]. However, Android IPCs are meanwhile significant attack vectors that can be leveraged to carry out attacks such as confused deputy and man-in-the-middle [92, 64, 76, 78].

While Android relies upon a tailored Linux environment, it still inherits a subset of traditional/native Linux IPCs (which are distinct from Android IPCs), such as signals, Netlink sockets, and Unix domain sockets. In fact, they are heavily utilized by the native layer of the Android runtime. Exposed Linux IPC channels, if not properly protected, could be abused by adversaries to exploit vulnerabilities within privileged system daemons and the kernel. Several vulnerabilities (*e.g.,* CVE-2011-1823, CVE-2011-3918, and CVE-2015-6841) have already been reported. Vendor customizations make things worse, as they expose additional Linux IPC channels: CVE-2013-4777 and CVE-2013-5933. Nevertheless, the use of Linux IPCs on Android has not yet been systematically studied.

In addition to the Android system, apps also have access to the Linux IPCs implemented within Android. Among them, Unix domain sockets are the only one apps can easily make use of: signals are not capable of carrying data and not suitable for bidirectional communications; Netlink sockets are geared for communications across the kernel space and the user space. The Android software development kit (SDK) provides developers Java APIs for using Unix domain sockets. Meanwhile, Android's native development kit (NDK) also provides native APIs for accessing low-level Linux features, including Unix domain sockets. Unix domain sockets are also known as local sockets, a term which we use interchangeably. They are completely different from the "local socket" in ScreenMilker [107], which refers to a TCP socket used for local IPC instead of network communication.

Many developers use Unix domain sockets in their apps, despite the fact that Google's

best practices encourage them to use Android IPCs [9]. The reason being Android IPCs are not suited to support communications between an app's Java and native processes/threads. While there are APIs available in SDK, no such API exists in the native layer. As a result, developers must resort to using Unix domain sockets to realize cross-layer IPC. Furthermore, some developers port existing Linux programs and libraries, which already utilize Unix domain sockets, to the Android platform.

Android IPCs are well documented on the official developer website, replete with training materials and examples. This helps educate developers on best practices and secure implementations. However, there is little documentation about Unix domain sockets, leaving developers to use them as they see fit — this may result in vulnerable implementations. Moreover, using Unix domain sockets securely requires expertise in both Linux's and Android's security models, which developers may not have.

Motivated by the above facts, we undertake the first systematic study focusing on the use of Unix domain sockets on Android. We present *SInspector*, a tool for automatically vetting apps and system daemons with the goal of discovering potential misuse of Unix domain sockets. Given a set of apps, SInspector first identifies ones that use Unix domain sockets based on API signatures and permissions. SInspector then filters out apps that use Unix domain sockets securely and thus are not vulnerable. We develop several techniques to achieve this, such as socket address analysis and authentication check detection. For system daemons, SInspector collects runtime information to assist static analysis. SInspector reports potentially vulnerable apps and system daemons for manual examination. We also categorize Unix domain socket usage, any security measures employed by existing apps and system daemons, and common mistakes made by developers. From this study, we suggest countermeasures in regard to OS-level changes and secure Unix domain socket IPC for both app and system developers. In this work, we do not consider network sockets, as local IPC is not their common usage.

We find that only 26.8% apps and 15% system daemons in our dataset enforce proper

security checks in order to prevent attacks exploiting Unix domain socket channels. All apps using a particular Unix domain socket namespace are vulnerable to at least DoS attacks. We uncover a number of serious vulnerabilities in apps. For example, we are able to gain root privilege by exploiting a popular root management tool, as well as grant/deny any other app's root access, without any user awareness. Moreover, we discover vulnerabilities with customizations on LG phones and daemons implemented by Qualcomm. These vulnerabilities allow us to factory reset the device, toggle the SIM card, and modify system date and time.

In summary, we make the following contributions in this work:

- We develop SInspector for analyzing apps and system daemons to discover potential vulnerabilities they expose through Unix domain socket channels. We overcome challenges in identifying socket addresses, detecting authentication checks, and performing data flow analysis on native code.

- Using SInspector, we perform the first study of Unix domain sockets on Android, including the categorization of usage, existing security measures being enforced, and common flaws and security implications. We analyze 14,644 apps and 60 system daemons, finding that 45 apps, as well as 9 system daemons, have vulnerabilities, some of which are very serious.

- We conduct an in-depth analysis of vulnerable apps and daemons that fail to properly protect Unix domain socket channels, and suggest countermeasures and better practices for utilizing Unix domain sockets.

6.2 Unix Domain Sockets

As we have mentioned in §2.3, Android claims that "Developers less familiar with security practices will be protected by safe defaults." [8] However, Unix domain sockets undermine this security philosophy. They are unable to achieve the same guarantees as well

as the Android IPCs. In particular, according to our analysis, Android APIs for using Unix domain sockets expose unprotected socket channels by default.

A Unix domain socket is a data communications endpoint for exchanging data between processes executing within the same host operating system. It supports the transmission of a reliable stream of bytes (`SOCK_STREAM`, similar to TCP). In addition, it supports ordered and reliable transmission of datagrams (`SOCK_SEQPACKET`), or unordered and unreliable transmission of datagrams (`SOCK_DGRAM`, similar to UDP).

Unix domain sockets differ from Internet sockets in that (1) rather than using an underlying network protocol, all communication occurs entirely within the operating system kernel; and (2) servers listen on addresses in Unix domain socket namespaces, instead of IP addresses with port numbers. Traditionally, there are two Unix domain socket address namespaces, as shown in Table 6.1.

Table 6.1: Unix domain socket namespaces.

Namespace	**Has socket file**	**Security enforcement**	
		SELinux	File permission
FILESYSTEM	YES	YES	YES
ABSTRACT	NO	YES	N/A

FILESYSTEM. An address in this namespace is associated with a file on the filesystem. When the server binds to an address (a file path), a socket file is automatically created. Socket file permissions are enforced by Linux's discretionary access control (DAC) system. The server must have the privilege to create the file with the given pathname, otherwise binding fails. Other processes who want to communicate with the server must have read-/write privileges for the socket file. By setting permissions of the socket file properly, the server can prevent unauthorized connections. The Android framework introduces a new namespace called *RESERVED*, which is essentially a sub-namespace of FILESYSTEM. Socket files are located in a dedicated directory, `/dev/socket/`, reserved for system use.

ABSTRACT. This namespace is completely independent of the filesystem. No file per-

missions can be applied to sockets under this namespace. In native code, an ABSTRACT socket address is distinguished from a FILESYSTEM socket by setting `sun_path[0]` to a null byte `'\0'`.

The Android framework provides APIs for using Unix domain sockets from both Java code and native code. These APIs use ABSTRACT as the default namespace, unless developers explicitly specify a preferred namespace. All Unix domain socket addresses are publicly accessible from file `/proc/net/unix/`. SELinux supports fine-grained access control for both FILESYSTEM and ABSTRACT sockets, so does SEAndroid. Compared to FILESYSTEM sockets, ABSTRACT sockets are less secure as DAC does not apply. However, they are more reliable; communication over a FILESYSTEM socket could be interrupted if the socket file is somehow deleted.

6.2.1 Threat Model and Assumptions

Unix domain sockets are designed for local communications only, which means the client and server processes must be on the same host OS. Therefore, they are inaccessible for remote network-based attackers. Our threat model assumes a malicious app that attempts to exploit exposed Unix domain socket channels is installed on the user device. This is realistic since calling Unix domain socket APIs only requires the `INTERNET` permission, which is so commonly used [3] that the attacker can easily repackage malicious payloads into popular apps and redistribute them. The attacker may also build a standalone exploit app which evades anti-malware products due to its perceived low privilege.

Table 6.2: Types of attacks by exploiting Unix domain sockets.

Role	Prerequisite(s)	Attacks
Malicous Server	1) Start running ahead of the real server 2) Client has no/weak authentication of server	Data Leakage/Injection, DoS
Malicous Client	Server has no/weak authentication of client	Privilege Escalation, Data Leakage/Injection, DoS

We summarize attacks malware can launch in Table 6.2. It is able to impersonate either

a client or a server to talk to the reciprocal host. A rogue Unix domain socket server could obtain sensitive data from clients or feed clients fake data to impact client functionality. A mock Unix domain socket client could access server data or leverage the server as a confused deputy [96]. In general, we classify a Unix domain socket as vulnerable if the server accepts valid commands through its socket channel without performing any authentication or similarly a client connects to a server without properly authenticating the server. This allows a nefarious user to retrieve sensitive information or access otherwise restricted resources through the Unix domain socket server/client it communicates with. Moreover, an ABSTRACT address can only be bound to by one thread/process. Apps using ABSTRACT namespace are vulnerable to DoS because their addresses could be occupied by malware.

6.3 Design and Implementation

The goal of SInspector is to examine the use of Unix domain socket in apps and system daemons, and identify those that are most likely vulnerable for validation. In this section, we describe our design and implementation of SInspector.

An ideal solution is to analyze all program paths in a program starting from the point of accepting a Unix domain socket connection, and then identify whether critical functions (end points) can be invoked without encountering any security checks. However, it is not practical for us to define a comprehensive list of end points and use dependencies between entry and end points to reason whether an app is vulnerable. First of all, apps may contain native libraries/executables, in which they make system calls to implement certain functionality, but there is no mapping between Android permissions and Linux system calls. It is imprecise to identify app behaviors based on system calls they make. Second, in our threat model, the malware runs on the same device as the vulnerable app/system daemon to be exploited, thus any data leaked from the target app/system daemon can possibly be a building block for more sophisticated attacks. However, it unknown to us which end points are potentially related to data leakage. More importantly, an incomplete list of end points

would result in significant false negatives.

Therefore, to evaluate which apps/system daemons are vulnerable, we choose to conservatively *filter out apps and system daemons that are definitely not vulnerable* (denoted by S_{nv}) — the others are considered to be potentially vulnerable (denoted by S_{pv}) — instead of directly identifying vulnerable apps. We have $S_{pv} = S - S_{nv}$, where S represents the whole set of apps/system daemons.

6.3.1 Our Approach

Due to the different characteristics of apps and system daemons, we adopt different techniques to analyze them. Figure 6.1 shows the modules and overall analysis steps of SInspector. Each step rules out a subset of apps/system daemons that are not vulnerable.

6.3.1.1 App Analysis

Given a set of apps, SInspector first employs *API-based Filter* to filter out those not using Unix domain sockets or having insufficient permission to use Unix domain sockets. Then, *Address Analyzer* finds out Unix domain socket address(es) each app uses, and discards apps whose addresses are under protection. They are not vulnerable because proper socket file permissions are able to prevent unauthorized accesses to a filesystem-based Unix domain socket channel. Next, the apps left are further examined by *Authentication Detector*. It detects and categorizes authentication mechanisms apps implement. Those adopting strong checks are considered to be not vulnerable. After that, *Reachability Analyzer* checks whether the vulnerable code that uses Unix domain socket will be executed or not at runtime. If not, that code is not reachable and will never be triggered, thus the app is not vulnerable. It ends up with a relatively small set of apps that are potentially vulnerable. Manual efforts are finally required to confirm the existence of vulnerabilities.

API-based Filter. This module filters out apps that are not in our analysis scope. For each app, it checks (1) Android permissions the app declares, (2) Java APIs the app calls, and (3)

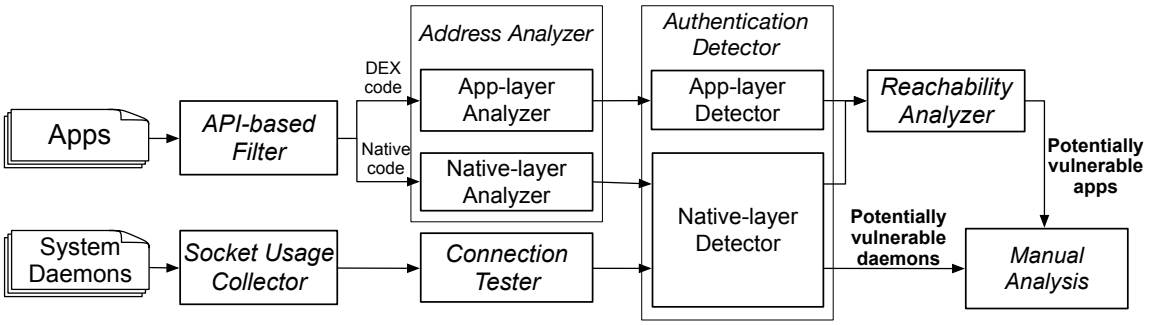

Figure 6.1: Overview of our approach to identifying potentially vulnerable apps and system daemons.

Linux system calls if the app has native code. Since using Unix domain sockets requires the `INTERNET` permission, apps without this permission are surely not vulnerable, neither are apps that do not invoke related APIs or system calls. APIs called through Java reflection are currently not considered, because (1) all socket APIs are available in Android SDK, unlike some private or hidden APIs which can only be called via Java reflection; and (2) Unix domain sockets just require a common, non-dangerous permission and therefore apps have little intention to hide the relevant logic.

Address Analyzer. This module identifies socket addresses each app uses and determines if their corresponding Unix domain socket channels are protected. Dalvik byte code and native code are analyzed by Address Analyzer's two submodules, *App-layer Analyzer* and *Native-layer Analyzer*, respectively.

Being aware of Unix domain socket address(es) an app connects to and/or listens on has two benefits. First, we can leverage addresses to determine if both client logic and server logic present in the same app. Usually, it is much easier to craft server exploits by replaying client behaviors, and vice versa. Second, different apps may use common libraries that utilize Unix domain sockets to implement certain functionality. We can take advantage of addresses to better group apps according to the libraries they use, because of the fact that apps using the same library typically have the same Unix socket address (or address structure). This is more reliable than identifying libraries merely based on package names and class names, as package names and class names could be easily obfuscated by

tools like ProGuard [34]. Though code similarity comparison techniques are also capable of recognizing libraries used across different apps, they are usually heavyweight.

Besides identifying addresses, Address Analyzer also evaluates whether the socket channel on an address is secure or not. As we have mentioned in §6.2, when using FILESYSTEM addresses, Unix domain socket servers are able to restrict client accesses by setting proper file permissions for socket files they listen on. A socket file satisfying the following conditions has proper permissions, and therefore the app using it is considered not vulnerable. First, it is located in the app's private data directory, *i.e.,* `/data/data/app.pkg.name/`. By default socket files created under this directory can only be accessed by the app itself. Second, there is no other operation altering the socket file's permissions to publicly accessible. The app, as the socket file's owner, has the privilege to change its permissions to whatever it wants. All file operations that possibly change the socket file's permissions need to be examined.

Authentication Detector. The OS allows both the client and the server to get their peers' identity information (*i.e.,* peer credentials) once a Unix domain socket connection is established. This module detects and categorizes authentication checks built on peer credentials. It also consists of two submodules for processing non-native and native code separately. Peer credentials are only available for Unix domain sockets. In our threat model, they are absolutely reliable because they are guaranteed by the kernel and therefore cannot be spoofed or altered by any non-root process in the user space. In Java code, apps call Android SDK API `LocalSocket.getPeerCredentails()` to get a socket peer's credentials, containing three fields: PID, UID, and GID. While in native code, the system call `getsockopt` is used to obtain the same information. Based on UID, GID and PID, servers and clients can implement various types of peer authentication mechanisms. Authentication Detector keeps track of the propagation of peer credentials in code, detects checks built upon the credentials, and categorizes them according to the particular credential they depend on. Peer authentication checks derived from UID and GID are considered

to be strong, as UID and GID are assigned by the OS and cannot be spoofed. However, authentications based on PID are relatively weak. Further analysis is unnecessary for apps employing strong checks.

Reachability Analyzer. The presence of Unix domain socket APIs in code does not necessarily mean the app actually uses Unix domain sockets at runtime. It is possible that the app just imports a library that offers functionality implemented with Unix domain sockets, but that part of code is never executed. To filter out such apps, Reachability Analyzer collects all possible entry points of an app, from which it builds an inter-component control flow graph. If Unix domain socket code cannot be reached from either of the entry points, we believe the code will not be reached at runtime, thus the app is considered not vulnerable.

6.3.1.2 System Daemon Analysis

Several obstacles make the pure static analysis of system daemons infeasible. First, given a factory image that contains all system files, it is difficult to extract all required data from it due to the fact that vendors develop their own file formats and there is no universal tool to unpack factory images. Second, different from apps, system daemons' Unix domain socket channels are usually enforced with specific SEAndroid policies made by Google or vendors. In this case, evaluating the security of a Unix domain socket channel becomes more complicated, especially for the FILESYSTEM namespace, because it is determined by both SEAndroid and socket file permissions.

However, system daemons are suitable for dynamic analysis without worrying about potential code coverage issues. They start automatically, serve as Unix domain socket servers waiting for client connections, and provide no user interface. It is reasonable to assume that their server logics are always running instead of being started on demand. Therefore, instead of employing API-based Filter and Address Analyzer, SInspector collects runtime information to find out system daemons using Unix domain socket with *Socket Usage Collector*, then test all socket channels daemons expose with *Connection Tester*, to

see which ones are accessible for an unprivileged app. The native layer Authentication Detector is reused for detecting and categorizing checks inside system daemons.

Socket Usage Collector. It is impossible for us to exploit vulnerable client logics implemented inside system daemons. One prerequisite of attacking client is being able to start running before the real server. In our threat model, however, the third-party app with only the `INTERNET` permission can never run ahead of a system daemon, which is started by the `init` process even before the Android runtime is initialized. Socket Usage Collector gathers runtime information of each Unix domain socket, including address, the process that listens on the address, protocol type (DGRAM, STREM, or SEQAPCKET), and corresponding system daemon.

Connection Tester. According to socket channel information collected, Connection Tester attempts to connect to them one by one, acting like a client running as a third-party app with `INTERNET` permission. If a socket channel is enforced by either file permissions or SEAndroid policies, the connection will be denied because of insufficient privilege. A system daemon is not vulnerable if all its socket channels are well protected.

6.3.1.3 Manual Analysis

For apps and system daemons that are most likely to be vulnerable, manual reverse engineering efforts are required to investigate the existence of vulnerabilities. Various tools are helpful for statically and dynamically reversing apps, *e.g.,* JEB [22], the Xposed framework [48], and IDA Pro. The effort needed for validating vulnerable code is supposed to be minimal, although writing workable exploits may take longer. Message formats (or called protocols) apps and system daemons use could be quite ad-hoc. Reverse engineering efforts largely depend on the complexity of implementation. In order to reduce human efforts, we could integrate protocol reversing techniques proposed in prior work [66, 77, 108] into SInspector in the future.

6.3.2 Implementation

We implement SInspector based on two cutting-edge tools, Amandroid [141] and IDA Pro. Both of them offer great extensibility and are friendly to plugin development. We take advantage of Amandroid to build inter-procedural control flow graph (ICFG), inter-procedural data flow graph (IDFG), and data dependence graph (DDG) from apps' non-native part for performing app-layer analysis, and leverage IDA Pro's disassembler and control flow analysis to build data flow analysis on native code, including apps' ELF libraries/executables and system daemons. SInspector only supports 32-bit ARM binaries for now, considering that the majority of Android devices are equipped with 32-bit ARM architecture processors.

Analyzing Apps. API-based Filter extracts `AndroidManifest.xml`, decodes it, and looks for the `INTERNET` permission. App code written in Java is compiled into one or more DEX files, in which all invoked APIs are visible. Native binaries are in ELF format. IDA Pro is able to identify direct system calls represented as constant relative addresses embedded in the instructions. However, it does not resolve indirect call targets that are stored in registers. More specifically, binaries can use the `SVC` instruction to do system calls, by specifying a system call number in register `R7` and then executing `SVC 0`. We extract the mapping between system call numbers and system call names from `arch/arm/include/asm/unistd.h` found in Android kernel 3.14, and identify all indirect system calls by inspecting `R7`'s values before each `SVC 0` instruction.

The app-layer of Address Analyzer and Authentication Detector are implemented on top of Amandroid. The server logic and the client logic are analyzed separately. We first locate the method in which Unix domain socket server/client is initialized, and create a customized entry point to it, then invoke Amandroid to build ICFG, IDFG, and DDG from the entry point. In Java code, Unix domain socket address is represented by the `LocalSocketAddress` class, whose constructors accept an address string as the first parameter. We look at construction sites of `LocalSocketAddress` objects.

In some cases, constant strings are used. In other cases where an address is built from package name, random integer, *etc.*, we track its construction of procedure by querying dependencies on DDG. Such an example is shown in Figure 6.2, in which we need to apply data flow analysis to extract the address as `"com.qihoo.socket"+System.currentTimeMillis()%65535`. This allows us to group apps that share the same socket address or have the same address construction procedure.

```
public static String getAddr() {
  return String.format("com.qihoo.socket%x",
    Long.valueOf(System.currentTimeMillis() & 65535));
}

protected void b(…) {
  …
  String addr = getAddr();
  this.serverSock = new LocalServerSocket(addr);
  …
}
```

Figure 6.2: A dynamically constructed socket address case.

The app-layer Authentication Detector finds paths on ICFG from `LocalServerSocket.accept()` (for server) and `LocalSocket.connect()` (for client) to `LocalSocket.getInputStream()` or `LocalSocket.getOutputStream()`. If we find that `LocalSocket.getPeerCredentials()` is called along the paths, and there is control dependency between either `getInputStream()`/`getOutputStream()` and `getPeerCredentials()`, authentication happens. In order to categorize authentication checks, we look at which fields (UID, GID or PID) are retrieved. We also define methods in `Context` and `PackageManager` that take UID, GID, or PID as sinks, and run taint analysis to track propagation paths. As mentioned in §6.3.1, checks relying on UID and GID are considered strong, while others are weak.

The native-layer Address Analyzer leverages intra-procedural control flow graph (CFG) generated by IDA Pro. Each basic block consists of a series of ARM assembly code disassembled by IDA Pro's state-of-the-art disassembling engine. We perform intra-procedural

data flow analysis on the CFG, following the classical static analysis approach [120]. Computing data flow at the assembly level is complicated, since we have to take into consideration both registers and the function stack. Unfortunately, there does not exist any robust tools that can perform data flow analysis on ARM binaries. ARM is a load-store architecture, and no instructions directly operate on values in memory. This means values must be loaded into registers in order to operate upon them. Therefore, we need to carefully handle all commonly used instructions that operate on registers and memory, especially load and store (pseudo) instructions. We examine the second argument of system calls `bind()` and `connect()`, which is an address pointing to the `sockadd_un` structure. Unix domain socket string is copied to the `sun_path` field, 2 byte off the start of `sockadd_un`. The first byte of `sun_path` indicates address namespace.

The native-layer Authentication Detector also performs intra-procedural data flow analysis. `getsockopt` has five parameters in total. Among them, the third one (option name) and the fourth one (option value pointer) are crucial. When option name is an integer equal to 17 (macro `SO_PEERCRED`), the option value will be populated by peer credentials, a structure consisting of three 4-byte integers: PID, UID, and GID. In other words, suppose option value's address is `A`, PID, UID, and GID will locate at addresses `A`, `A+4`, and `A+8`, respectively. When `getsockopt` is called, we inspect option name and record option value's address on the stack `A`. After that, functions that access values at `A`, `A+4`, or `A+8` are considered as checks.

Analyzing System Daemons. Socket Usage Collector calls a command line tool `netstat` to get interested socket information. Note that the default `netstat` shipped with Android has very limited capability. We choose to install `busybox`, which provides a much more powerful `netstat` applet. Root access of the Android device is required, otherwise `netstat` will not be able to find out the process that listens on a particular socket address. We build Connection Tester into a third-party app that requests only the `INTERNET` permission. Native-layer Authentication Detector is reused for analyzing sys-

tem daemons.

6.3.3 Limitations

One limitation of SInspector is that we have to rely on human efforts to generate exploits. Even though we can find out apps and system daemons that are highly likely to be vulnerable, we are not able to automatically craft exploits to finally validate vulnerabilities. SInspector may have false positives, because of our conservative strategies for filtering out insusceptible apps and system daemons. The native-layer intra-procedural data flow analysis is likely to miss data flows across different functions.

We may also have false negatives: (1) we cannot handle dynamically loaded code; and (2) native executables/libraries might be packed or encrypted. They could introduce uncaught control and data flows.

6.4 Results

We evaluate SInspector with a total number of 14,644 up-to-date Google Play applications crawled in mid-April 2016, including approximately top 340 from all 44 categories. Google has imposed restrictions to ensure that applications can only be downloaded through the Google Play on user devices, which makes it difficult for us to obtain APK files. To tackle this, we crawl meta data of applications (*e.g.,* package name, version name) from Google Play and download corresponding APK files from ApkPure [10], a mirror of Google Play that allows free downloading.

We also use three phones to evaluate SInspector: (1) LG G3 running Android 4.4.4, (2) Samsung Galaxy S4 running Android 5.0.1, and (3) LG Nexus 4 running 5.1.1. All of them are updated to the latest firmware and rooted. Most of the recently released Android phones either are equipped with 64-bit ARM processors or cannot be rooted. They are not suitable for our experiments because SInspector's dynamic analysis requires root access and the static data flow analysis can only handle 32-bit ARM binaries.

6.4.1 Overview

Table 6.3 shows the overall statistics on Unix domain socket usage among applications and system daemons. Application data are from API-based Filter and daemon data come from Socket Usage Collector. Among 14,644 applications, 3,734 (25.5%) have Unix domain socket related APIs or system calls in code, and the majority of them (3,689) use ABSTRACT addresses, while only a few use FILESYSTEM and RESERVED addresses. The sum of numbers in each address namespace may be greater than the total number, as an application or a system daemon could use more than one namespaces.

Different from applications, most of the system daemons use RESERVED addresses. Compared to Nexus 4 running non-customized Android, LG G3 and Galaxy S4 have more system daemons and heavier usage of ABSTRACT addresses. This fact clearly shows that vendor customizations inevitably expose more attack vectors.

Table 6.3: Numbers of applications/system daemons that use Unix domain sockets.

	# Applications	**# Daemons**		
		LG G3	Galaxy S4	Nexus 4
ABSTRACT	3,689	5	8	2
FILESYSTEM	36	4	5	2
RESERVED	20	13	17	11
Total	3,734	20	27	13

6.4.1.1 Libraries

We summarize identified libraries utilizing Unix domain sockets in Table 6.4. "Singleton" and "Global lock" in the Usage column will be described later in §6.4.2. We observe that 3,406 applications use an outdated Google Mobile Services (GMS) library alone and exclude them. The outdated GMS library is potentially vulnerable to DoS and data injection attacks. The latest GMS library has completely discarded Unix domain sockets, which implies that Google may have been aware of potential problems of using Unix domain sockets. Except for Amazon Whisperlink and OpenVPN, all other libraries use the

ABSTRACT namespace, making them all vulnerable to DoS.

6.4.1.2 Tool Effectiveness and Performance

Besides applications using common libraries listed in Table 6.4, SInspector found 73 potentially vulnerable applications having no authentication or weak authentications. Table 6.5 summarizes analysis effectiveness. After reachability analysis, SInspector finally reported 67 applications that are most likely to be vulnerable. We manually looked at all 67 applications and confirmed that 45 are indeed vulnerable. SInspector reported 12 potentially vulnerable system daemons. After manual examination, we confirmed 9 of them are truly vulnerable. We present a case study of most critical vulnerabilities in §6.5.

Table 6.4: Libraries that use Unix domain socket.

Library	**# Applications (reachable)**	**Usage**	**Namespace[1]**	**Auth**	**Susceptible attack(s)[2]**
Baidu Push	9 (9)	Singleton	ABS	N/A	DoS
Tencent XG	11 (11)	Singleton	ABS	N/A	DoS
Umeng Message	17 (17)	Singleton	ABS	N/A	DoS
Facebook SocketLock	13 (13)	Global lock	ABS	N/A	DoS
Yandex Metrica	95 (95)	Global lock	ABS	N/A	DoS
Facebook Stetho	97 (97)	Debugging interface	ABS	Permission	DoS
Sony Liveware	8 (5)	Data transfer	ABS	None	DoS, DI, DL
Samsung SDK	12 (10)	Data transfer	ABS	None	DoS, DI, DL
QT5	10 (10)	Debugging interface	ABS	None	DoS, DI, DL
Clean Master	9 (9)	Data transfer	ABS	None	DoS, DI, DL
Amazon Whisperlink	11 (7)	Data transfer	FS	None	N/A
OpenVPN	7 (4)	Cmd & control	FS	None	N/A

[1] ABS and FS in this column are short for ABSTRACT and FILESYSTEM.
[2] DI and DL stand for data injection and data leakage.

All experiments are done on a machine with 3.26GHz $\times$ 8 Core i7 and 16GB of memory. The most compute-intensive module of application analysis is Reachability Analyzer. Depending on the numbers of bytecode instructions of applications, Reachability Analyzer

Table 6.5: SInspector results summary.

	Potentially vulnerable	True positive	False positive	Precision
Applications	67	45	22	67.2%
LG G3	6	4	2	66.7%
Galaxy S4	5	4	1	80%
Nexus 4	1	1	0	100%

could take a few minutes to more than one hour. Other modules are pretty fast. The average time for analyzing one application is 2,502 seconds. For system daemon analysis, IDA Pro's disassembling process took a few seconds to a few minutes, the average time for analyzing a system daemon is 39 seconds.

6.4.2 Unix Domain Socket Usage

Unix domain sockets provide a means to perform IPC, but it turns out the usage in the wild is not limited to IPC. According to our experience in inspecting potentially vulnerable applications SInspector reported, we extract code patterns for categorizing Unix domain socket usage and summarize them in Table 6.6. We observe that Unix domain sockets are widely used by applications to implement global locks and singleton, as well as to implement watchdogs.

Table 6.6: Code patterns for categorizing Unix domain socket usage.

Usage	Key APIs	Code pattern	# Applications

IPC	`LocalSocketServer.<init>()`
	`LocalSocketServer.accept()`
	`LocalSocket.connect()`
	`LocalSocket.getInputStream()`
	`LocalSocket.getOutputStream()`
		Unix domain socket server/client reads data from (or write data to) the other end.	193

Singleon/ Global Lock	`LocalServerSocket.<init>()` `LocalSocket.bind()`	Server has no reading/writing operations after binding to an address.	165
Watchdog	`LocalSocket.connect()` `LocalSocket.getInputStream()`	Client connects to server and then blocks at reading. Server also blocks at reading after accepting client connection.	33

6.4.2.1 Inter-Process Communication

Not surprisingly, the prominent usage of Unix domain sockets is performing IPC. Applications are free to implement their own protocols for client/server communication. However, we do find a very unique use of Unix domain socket as an IPC mechanism. A few video recording applications leverage Unix domain sockets to realize real-time media streaming, a feature that Android's media recording APIs do not support. Developers came up with a workaround, which takes advantage of an existing media recording API `setOutputFile(fd)` that outputs camera and microphone data stream to a file descriptor. After a Unix domain socket connection is established, the client passes its output file descriptor to this API so that the server can read real-time camera/microphone output. In this way, media output is converted to a stream that can be further processed in real time, *e.g.,* to perform live streaming.

6.4.2.2 Realizing Singleton

An ABSTRACT socket address can only be bound on by one Unix domain socket server instance. Once an address has been taken, another server that attempts to bind on it would fail. This feature is widely exploited to ensure that certain code will not be executed more than once. In fact, the `PhoneFactory` class in AOSP "use UNIX domain socket to prevent subsequent initialization" of the `Phone` instance, as Figure 6.3 shows.

```
105 try {
106   // use UNIX domain socket to
107   // prevent subsequent initialization
108   new LocalServerSocket("com.android.internal.telephony");
109 } catch (java.io.IOException ex) {
110   hasException = true;
111 }
```

Figure 6.3: `com.android.internal.telephony.PhoneFactory` uses a Unix domain socket for locking. Code excerpted from AOSP 6.0.1_r10.

Baidu Push, Tencent XG, and Umeng Message are three top message push service

providers in China. Due to the state-level blocking of Google services, Google Cloud Messaging (GCM) is not accessible. Therefore, applications targeting on China market have to choose other push services. It is likely that multiple applications integrated the same push service library co-exist on the same device. That would be less power-efficient if they each run their own push service. They choose to share one push service instance across multiple applications and realize that with a Unix domain socket.

6.4.2.3 Implementing Global Lock

This use case also takes advantage of the feature that ABSTRACT addresses are used exclusively. There is demand on global locks because some resources cannot be used by two different processes/threads simultaneously, or certain operations should be serialized instead of parallelized. However, Android itself does not provide global locks shared between different applications. Facebook applications all have a DEX optimization service. They will not do optimization before successfully acquiring a global lock implemented with a Unix domain socket. This ensures that only one optimization task runs in the background, and helps reduce the negative impact on user experience.

6.4.2.4 Implementing Watchdog

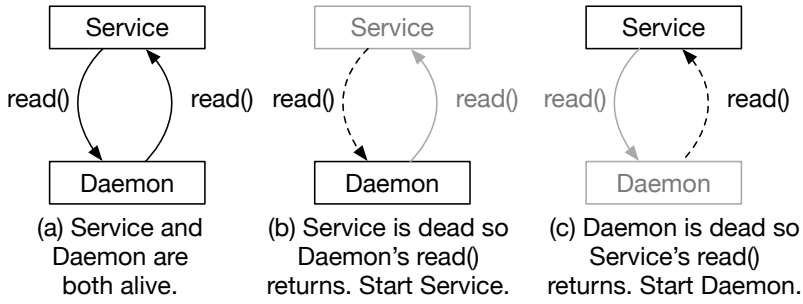

Figure 6.4: The Kaspersky application's service and daemon monitor each other through a Unix domain socket.

Some applications have important services that are expected to always run in the background. Such "immortal" services are against Android's memory management philosophy,

and therefore developers have to to find a workaround to automatically restart them, in case they are somehow terminated. They implement a watchdog mechanism leveraging Unix domain sockets. For example, the Kaspersky Security application starts a native daemon in a service. The daemon and the service monitor each other mutually, through a Unix domain socket channel. If one has died, the other will gets notified and restart it immediately, as Figure 6.4 depicts.

6.4.3 Peer Authentication

We refine the categorization made by SInspector's Authentication Detector module, and classify peer authentication checks into four types: UID/GID checks, process name checks, user name checks, and permission checks. Table 6.7 shows the numbers of applications and system daemons adopting each type of checks. Applications and daemons tend to use different types of authentication checks. Applications only adopt UID/GID checks and permission checks, while system daemons use all checks except permission checks. One possible reason is in different layers the information applications/system daemons can obtain differs. On the application layer, applications can easily get the peer application's permissions with its UID. However, there are no APIs for getting the peer's process name or user name. On the native layer, process name and user name can be easily obtained. But due to the lack of Android runtime context, it is infeasible to query the peer's permissions. Only 9 of 60 (15%) daemons employ strong checks, meaning that their security heavily relies on the correctness of SEAndroid policies and file access permissions.

Table 6.7: Statistics on peer authenication checks.

	UID/GID	**Process name**	**User name**	**Permission**
#Applications	20	0	0	97
#Daemons	7	3	2	0

Process Name Checks. In native layer, getting process name with its PID is done by reading `/proc/PID/cmdline` or `/proc/PID/comm` on the proc filesystem (procfs).

Process name checks compare the peer's process name with predefined process name(s). By default, the process name of an Android application is its package name. Therefore, the content of the two proc files of an application process is actually the application's package name. Interestingly, we find that applications are able to modify their own process names at runtime, by calling a hidden method `Process.setArgV0(String s)` through Java reflection. This method is supposed to be used by the system (labeled with `@hide` in source code), but it requires no permissions. This hidden method makes all process name checks meaningless, as malicious applications can always change their process names to legitimate ones so that they can bypass checks and send messages to the victim. For example, the system daemon `cnd` on LG G3 and Galaxy S4 is used for managing Qualcomm connectivity engine [35]. It accepts requests from clients through a Unix domain socket and checks if the client's process name is "android.browser". Requests from other clients are not legitimate and will be discarded. By changing process name to "android.browser", any application can send legitimate requests to `cnd` effortlessly.

UID/GID Checks. Android reserves UIDs less than 10,000 for privileged users. For instance, the user `system` has 1,000 as both UID and GID. Normally, each application has its own UID and GID, but applications from the same developer could share the same UID and GID. These checks are handy when one wants to allow only privileged users or particular applications to communicate with it. UID/GID checks efficiently prevent unauthorized peers, as UID and GID can never be spoofed or modified. For example, the Android Wear application has a service called AdbHubService, which is used for remote debugging. It starts a Unix domain socket server accepting debugging commands from ADB shell. Only commands coming from `root` and `system` are allowed, by checking if a client's UID is equal to 0 or 2,000.

User Name Checks. These checks are similar to UID/GID checks, since each user also has its unique user name that cannot be spoofed or modified. They also effectively authenticate the peer's identity. Samsung Galaxy S4's RIL daemon, `rild`, checks client user name. A

list of names of privileged users is hardcoded in the binary, *e.g.,* media, radio. User name checks might be better than UID/GID checks because the same user may have different UID/GID on different devices due to vendor customization.

Permission Checks. These checks enforce that only applications with specific permissions can access the Unix domain socket channel. On the application layer, applications can call several APIs in the `Context` class to check another application's permissions. The Facebook Stetho library checks if the peer has the `DUMP` permission, a system permission that can only be acquired by system applications. It first obtains UID and PID from peer credentials, then calls `Context.checkPermission(permission, pid, uid)` to do permission checking

Token-Based Checks. Besides the aforementioned peer authentication checks, we observe two applications adopt token-based checks. The server and the client first securely share a small chunk of data (called *token*). The server compares the token of the incoming client with its own copy so that only clients having the right token can talk to it. This type of checks, assuming the token is shared in secure ways, can effectively prevent unauthorized accesses. We find two applications employing two different methods to share tokens between the server and the client. The first one, Helium Backup, broadcasts the token on the server side. The broadcast is protected by a developer-defined permission, and therefore other applications without the required permission cannot receive the token. The second one, OS Monitor, stores its token in a private file. Since the server and the client are both created by the application itself, they have privileges to read the private file and extract the token. SInspector currently cannot identify such checks. As a result, these applications reported as potentially vulnerable are actually false positives.

6.5 Case Study

By examining the output of SInspector, we successfully discovered several high-severity zero-day vulnerabilities affecting popular applications installed by hundreds of

millions of users, widely used third-party libraries, and system daemons having root privileges. These vulnerabilities can be exploited to (1) grant root access to any applications, giving the attacker entire control of the device, (2) read and write arbitrary files, allowing the attacker to steal user privacy and modify system settings, (3) factory reset the victim device, causing permanent data loss, and (4) change system date and time, resulting in denial of service.

6.5.1 Applications

6.5.1.1 Data Injection in a Rooting Tool

As rooting gaining popular in the Android community, many one-click rooting tools become available [145], which allow users to gain root access very easily. One major rooting tool, which claims to be able to root 103,790 different models (as of May, 2016), support a wide range of devices running Android 2.3 Gingerbread and above up to Android 6.0 Marshmallow. As well as rooting, the tool also serves as a root access management portal, through which users can grant or deny applications' root requests.

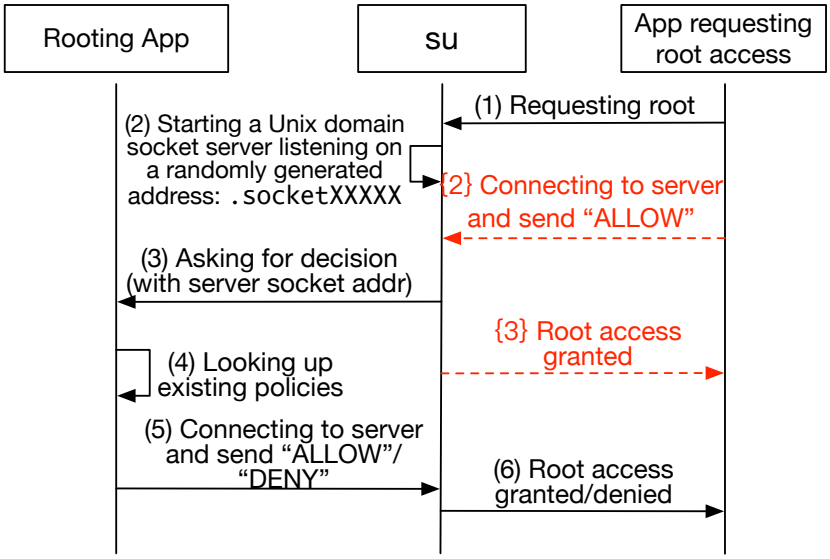

Figure 6.5: KingRoot vulnerability illustration.

Once a device is successfully rooted, the rooting application installs a command line

tool, `su`, to the system partition `/system/bin/su`. Applications then request root access by executing `su`, who starts a Unix domain socket server waiting for the rooting application to send back user decision. The rooting application looks up existing policies. If no corresponding policy exists, it pops up a dialog that asks the user to make a decision. Figure 6.5 illustrates the whole process. The standard root request procedure consists of steps (1)-(6) with solid arrow lines. By injecting an "ALLOW" string, any application can get root access regardless what the user's actual decision is, shown as steps (1){2}{3}.

However, the FILESYSTEM-based socket channel is publicly accessible as its file permissions are set to `rwxrwxrwx`, and there is no client authentication in `su`. As a result, any application can inject arbitrary decisions before the rooting application sends out the real decision to `su`. This allows a malicious application to grant or deny root access of any other applications, as well as grant itself root privileges in order to take full control of the device. We reported this vulnerability to the developers and they rated it as the most severe security bug in their product to date. They fixed the vulnerability and released a new version in 24 hours.

6.5.1.2 Privilege Escalation in ES File Explorer

ES File Explorer is a very popular file management application on Android, accumulating over 300 million installs. To perform file operations that Java layer APIs do not efficiently support, the application starts a native process and executes a binary, `libestool2.so`[1], which creates a Unix domain socket server listening on an ABSTRACT address, `@/data/data/com.estrongs.android.pop/files/comm/tool_port`. Moreover, if the device is rooted and the user chooses to run ES File Explorer in root mode, it starts another `libestools2.so` process with root privileges, listening on another ABSTRACT address, `@/data/data/com.estrongs.android.pop/files/comm/su_port`. Some low-level file operations, such as

[1]This binary looks like a shared library from its name, but it is essentially an ELF executable.

modifying file access permissions and changing file status/ownership are sent to these two native processes to execute.

Since there is no client authentication on the server side (*i.e.,* `libestool2.so`), any application can send the native processes commands to run. We were able to read an arbitrary application's private files and protected system files by exploiting this vulnerability, after successfully reversing the communication protocol between the ES File Explorer application and its native processes. This critical vulnerability was fixed two months after we first reported it to the developers.

6.5.1.3 DoS VPN Applications

Multiple OpenVPN clients for Android are available. *OpenVPN for Android* is an open source client that targets advanced users and offers many settings and the ability to import profiles from files and to configure/change profiles inside the application. The client is an ELF executable ported from the community version of OpenVPN for Linux.

OpenVPN management interface allows OpenVPN clients to be administratively controlled from an external program via a TCP or Unix domain socket. Quite a few of applications making use of OpenVPN for Android utilize Unix domain sockets to communicate with the management application. However, some of them fail to set file permissions correctly for the socket file. OpenVPN supports various client authentication mechanisms. Surprisingly, none of these applications adopt any client authentication. Consequently, an adversary can establish a connection to the management interface and then control the OpenVPN client, causing deny-of-service at least.

6.5.2 System Daemons

6.5.2.1 LG AT daemon

The privileged *AT Daemon*, `/system/bin/atd`, on (at least) the LG G3 is vulnerable, which allows any application with only the `INTERNET` permission to factory reset the

phone, toggle the SIM card, and more, causing permanent data loss and denial of service. `atd` is a proprietary daemon developed by LG. It starts a Unix domain socket server that performs no client authentication, listening on socket file `/dev/socket/atd`, whose permissions are not correctly configured (*i.e.,* `srw-rw---- system inet`). The permission configuration means all users in the `inet` Linux group can read and write this socket file. Android applications having the `INTERNET` permission all belong to the `inet` group. As a result, they are able to read and write this socket file so that they can talk to the AT daemon through this Unix domain socket channel. Commands from any applications, if in the right format, will be processed by the daemon.

By reversing the messages `atd` accepts, we figured out the format and successfully crafted commands that instruct `atd` to (1) perform factory reset, wiping all user data and (2) toggle the SIM card. In fact, `atd` accepts a large set of commands (only a subset were successfully reversed); reverse engineering the whole protocol would allow us to send arbitrary SMS requests, make phone calls, get user's geographic location, and so on.

6.5.2.2 Qualcomm Time Daemon

We first found that a LG G3 daemon `/system/bin/time_daemon` opens a Unix domain socket server listening on an ABSTRACT address `@time_genoff`. This daemon verifies the client's identity. However, the verification logic is very weak. It only checks whether the process name of the client is a constant string "comm.timeservice" and therefore can be easily bypassed.

This vulnerability allows any application with the `INTERNET` permission to change the system date and time, affording attackers to DoS services relying on exact system date and time, e.g., validating the server certificate. `/system/bin/time_daemon` is developed by Qualcomm, and other Android devices using Qualcomm time daemon are also vulnerable. This vulnerability has been reported and was assigned CVE-2016-3683.

6.5.2.3 Bluedroid

The Android Bluetooth stack implementation is called bluedroid, which exposes a Unix domain socket channel for controlling the A2DP protocol [1]. The ABSTRACT address, `@/data/misc/bluedroid/.a2dp_ctrl`, is expected to be enforced by SEAndroid. To our surprise, we are able to connect to the server through this address and send control commands to it on a Nexus 4. We are able to control the audio playing on a peripheral device connected to the phone through Bluetooth. Though the LG G3 and the Galaxy S4 also expose the same channel, accesses from third-party applications always fail at connecting stage due to insufficient permission. This case suggests that vendors may have made some security improvements despite their tendency to introduce vulnerabilities [148].

6.6 Countermeasure Discussion

As our study suggests, the misuse of Unix domain sockets on Android has resulted in severe vulnerabilities. We discuss possible countermeasures to minimize the problem from two aspects: (1) OS-level mitigations and (2) better approaches to implementing secure IPC that utilizes Unix domain sockets.

6.6.1 OS-Level Solutions

Changing the default namespace. For now, Unix domain socket channels created by applications use the ABSTRACT namespace by default. Due to the lack of DAC, socket channels based on ABSTRACT addresses are less secure than those based on FILESYSTEM addresses. Therefore, intuitive mitigation is to change the default namespace from ABSTRACT to FILESYSTEM; or more radically, disable the use of ABSTRACT namespace.

More fine-grained SEAndroid policies and domain assignment. In the current SEAndroid model, all third-party applications, although having individual UIDs and GIDs, are

assigned the same *domain* label, *i.e.,* `untrusted_app`. Unix domain sockets accesses between third-party applications are not enforceable by SEAndroid because domain-level policies cannot tell one third-party application from another.

Therefore, we need to assign different *domain* labels to different third-party applications so that more fine-grained policies can be made to regulate Unix domain socket accesses. Nevertheless, this could introduce new problems: pre-defined policies would not be able to cover applications, and making fixed policies editable at runtime may open new attack vectors. Moreover, it would be untenable to define policies for every application; each user may install any number of applications.

6.6.2 Secure IPC on Unix Domain Sockets

We demonstrate three scenarios where applications and system daemons require Unix domain sockets for IPC and discuss possible solutions to their security problems.

A privileged system daemon exposes its functionality to applications. A system daemon may need to provide diverse functionality to applications that have different privileges. For example, the LG AT daemon may want to expose the capability of doing factory reset to only system applications, and allow applications with location permissions to get the user's GPS coordinates. To achieve this, system daemons will have to enforce application permissions themselves. Unfortunately, the lack of Android runtime context in system daemons precludes daemons from easily obtaining the application's permission(s).

Figure 6.6 demonstrates the proposed solution. The goal is to delegate peer authentication to the existing Android security model. Instead of letting applications and daemons communicate directly through a Unix domain socket, a system service acts as an intermediary between the two. This new system service runs as the `system` user with UID 1000, thus can be easily authenticated by the daemon. Applications talk to this system service through Android Binder and their permissions are validated by the system service. In this way, daemon functionality is indirectly exposed to applications with the help of a system

service.

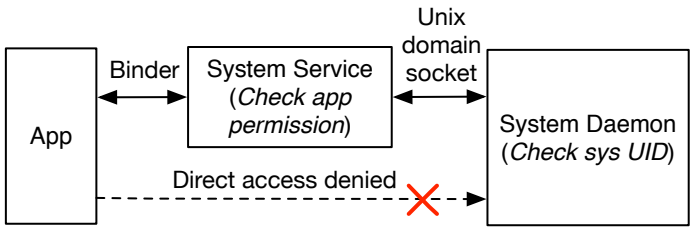

Figure 6.6: A secure way to expose system daemon functionality to applications. A system service is added between applications and the system daemon.

An application consisting of both Java and native code performs cross-layer IPC. Applications having native executables need an intra-application, cross-layer IPC. An application creates a native process to run its executable, and uses Unix domain sockets to communicate with the native process from its non-native part. In this case, executables still have the same UID as their owner applications. Therefore, it is convenient to check UID on both client and server sides.

An application exposes interfaces to other applications. Android-specific IPCs such as Intents are expected to be used for inter-application communications. However, applications have to choose Unix domain sockets for cross-layer IPCs. We propose a token-based mechanism inspired by Helium described in §6.4.3, as Figure 6.7 illustrates. The client application first sends a broadcast to the server application to request a communication token. The server responds by asking the user to allow or deny the incoming request. If the user allows, the server application generates a one-time token for that particular client and returns the token. After that, the client connects to the server with its token and a Unix domain socket connection will be established. Note that the token is not meaningful to anyone else. Even if it was stolen, the attacker would not be able to use it to talk to the server application.

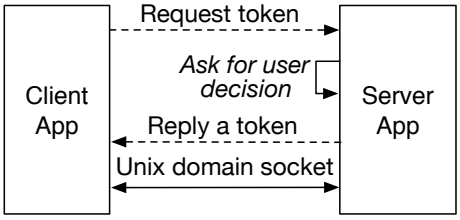

Figure 6.7: Token-based secure Unix domain socket IPC. Dotted arrow lines stand for permission-protected broadcasts.

6.7 Conclusion

In this paper, we conducted the first systematic study in understanding the usage of Unix domain sockets by both apps and system daemons as an IPC mechanism on Android, especially for cross-layer communications between the Java and the native layers. We presented SInspector, a tool for discovering potential security vulnerabilities through the process of identifying socket addresses, detecting authentication checks, and performing data flow analysis on native code. We analyzed 14,644 Android apps and 60 system daemons, finding that some apps, as well as certain system daemons, suffer from serious vulnerabilities, including root privilege escalation, arbitrary file access, and factory resetting. Based on our study, we proposed countermeasures to prevent these attacks from occurring.

CHAPTER VII

Conclusion and Future Work

7.1 Lessons Learned

Our work has revealed security and safety problems of exposed interfaces resulted from design, implementation, and configuration stages. We have learned several important lessons throughout this research. First, system designers cannot anticipate all threats and abuses. Diehard apps are a perfect example. In addition to vulnerabilities that have a direct impact on system security and user privacy, they should also take into account possible abuses during the design process. Since threat models are changing over time, they need to more proactively detect threats and learn from attacks. Second, more efforts should be invested in order to make sure the implementation faithfully realizes the design. We have seen that implementation flaws never be completely eliminated, because developers keep making mistakes that seem simple and avoidable. Advanced tools that detect implementation flaws at an early stage are very helpful for preventing abuses proactively. Third, making correct, secure configurations is challenging for developers and users. The "security by default" principle should always be followed. Automated tools for detecting insecure configurations and for generating secure configurations are useful, but domain knowledge is required to build such tools.

However, there are fundamental limitations in today's access control based protection of exposed interfaces. The prominent one is the lack of context. For instance, a malicious

application can disguise as a voice recorder and provide fully-fledged recording functions. The user, who occasionally uses the application, is highly likely to grant it microphone access. As a result, the malicious app can then abuse the microphone access and record user conversation stealthily at any time. There is a mismatch between user expectation and application behaviors: users expect applications to complete specific tasks, which oftentimes require privileges; but once access is granted, users have no control over applications' future use of the access.

In addition to specific defense and mitigation solutions for the concrete problems we have studied, we discuss generic design, implementation, and configuration improvements that can better prevent exposed interfaces from capability abuse.

- *Design: mandatory access control could be applied to mobile and safety-critical cyber-physical systems.* Discretionary access control lets users manage the content they own. Besides Linux file permissions, it is also good to let users of an online social network choose who accesses their data. However, mobile systems having abundant user privacy and safety-critical systems such as ICS require more rigorous control instead of flexibility. For instance, iOS leverages the TrustedBSD MAC framework to run applications in sandboxes. Android, however, only applies the SE-Android MAC framework in the kernel, leaving the middleware layer less strictly protected.

- *Implementation: centralized security enforcement can be better than distributed checks.* To regulate accesses to capabilities and sensitive resources, there are in general two strategies to implement enforcement: (1) centralized enforcement that examines all incoming access requests, and (2) distributed checks placed behind each exposed interface. We believe that centralized enforcement is more reliable and the implementation is less error-prone. Due to a large number of exposed interfaces, placing distributed checks requires much more developer efforts and security expertise. Nevertheless, the trade-off is efficiency. Distributed checks, if implemented

correctly, can reject unauthorized accesses much earlier. In contrast, the execution has to reach the centralized enforcement before being granted or denied. Many CPU times will be wasted if the access is eventually denied.

- *Configuration: context can be used as a new dimension for access control.* Prior work [101] has proposed to enforce contexts as execution paths at inter-procedure control and data flow levels. Our lightweight application lifecycle control framework also provides fine-grained event contexts. However, additional context information brings more complexity to policy making. Keeping users in the loop is not the optimal choice, because normal users may have difficulties in understanding contexts and making a wise decision. We believe that for a system that supports fine-grained contexts, machine learning techniques could be helpful to come up with customized policies for individual users.

7.2 Conclusion

This dissertation is dedicated to studying exposed system interfaces that can lead to abuses of system capabilities. In addition to systematically discovering vulnerabilities, this study helps understand the root causes in system design and implementation phases, and proposes mitigation solutions that are fundamental to preventing capability abuses. We have addresses four problems in this dissertation, spanning from popular smartphone systems that have complicated software stack to industrial control systems where rich interactions between software and physical parts exist.

First, we detect inconsistencies in access control policy enforcement in the Android framework. We design and build a tool that compares permissions enforced on different code paths and identifies the paths enforcing weaker or no permissions. Our methodology does not require security policies, which are non-trivial to learn, and it targets only on the enforcement. We have conservatively discovered at least 14 inconsistent security

enforcement cases—all officially confirmed—that can lead to security check circumvention vulnerabilities across important and popular services such as the SMS service and the Wi-Fi service, incurring impacts such as privilege escalation, denial of service, and soft reboot. Our findings also provide useful insights on how to proactively prevent such security enforcement inconsistency within Android.

Second, we propose the Application Lifecycle Graph (ALG), a novel modeling approach to describing system-wide app lifecycle. We develop a lightweight runtime framework that utilizes ALG to realize fine-grained app lifecycle control, with a focus on restricting diehard apps that abuse entry points to automatically start up and game the priority-based memory management mechanism to evade being killed. The framework exposes APIs that provide ALG information and lifecycle control capabilities to developers and device vendors, empowering them to leverage the framework to implement rich functionalities. Evaluation results show that the proposed framework is competent and incurs low performance overhead. It introduces 4.5MB additional memory usage on average, and approximately 5% and 0.2% CPU usage during system booting and at idle state.

Third, we study real-world programmable logic controller programs for identifying insecure configurations that can lead to critical security and safety violations. Our results show that it is common that PLC programs' tags have incorrectly configured visibility and accessibility. Among 433 complete programs, we identify 176 critical paths. Add-on instructions (AOIs) are similar to software libraries that can be reused by different programs. We find 117 out of 388 AOIs directly control hardware outputs, but no safety violation checks are employed. For example, conveyor speed can be changed externally by exploiting an AOI.

Lastly, we conduct the first systematic study in understanding the security properties of the usage of Unix domain sockets by both Android apps and system daemons as IPC channels, especially for cross-layer communications between the Java and the native layers. Our in-depth analysis revealed some serious vulnerabilities in popular apps and system

daemons, such as root privilege escalation and arbitrary file access. Based on our findings, we propose countermeasures and improved practices for securely using Unix domain sockets on Android.

7.3 Future Work

A number of future directions are worth exploring. We summarize them as follows.

- *Automated approaches to security policy enforcement.* As the complexity of a system grows, it becomes more error-prone for developers to manually add security checks. Since humans tend to make mistakes, no matter how competent they are, automated approaches may be more favorable. Recent research has proposed to use deep learning to automate vulnerability detection in source code [128] and triage bugs [111], as well as apply neural machine translation to learning bug-fixing patches [134]. More research effort is desired in order to enable bug-free security enforcement with automated tools.

- *Connected and autonomous vehicle (CAV) security.* Security of CAVs is a growing concern, first, due to the increased exposure of the functionality to the potential attackers; second, due to the reliance of car functionalities on diverse CAV systems; third, due to the interaction of a single vehicle with myriads of other smart systems in an urban traffic infrastructure [71]. A systematic security study on CAV systems, especially the interfaces exposed by different system components, is required to understand potential security and safety threats and to improve CAV security.

- *Critical infrastructure security.* Critical infrastructure describes the physical and cyber systems and assets that are vital to nations and people. Their incapacity or destruction would have a debilitating impact on the physical or economic security or public health or safety. There are much work can be done to understand what interfaces are exposed to attackers and what the security implications are.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Advanced audio distribution profile (a2dp). https://developer.bluetooth.org/TechnologyOverview/Pages/A2DP.aspx.

[2] Advanced task manager. https://play.google.com/store/apps/details?id=mobi.infolife.taskmanager.

[3] An Analysis of Android App Permissions. http://www.pewinternet.org/2015/11/10/an-analysis-of-android-app-permissions/.

[4] Android authority forums. https://www.androidauthority.com/community/.

[5] Android Binder. https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf.

[6] Android forums. https://androidforums.com/.

[7] Android forums at androidcentral. https://forums.androidcentral.com/.

[8] Android Security Overview. https://source.android.com/security/.

[9] Android Security Tips: Using Interprocess Communication. http://developer.android.com/training/articles/security-tips.html#IPC.

[10] ApkPure website. https://apkpure.com/.

[11] App programming guide for ios — background execution. https://goo.gl/jryM9q.

[12] Application fundamentals. https://developer.android.com/guide/components/fundamentals.html.

[13] ART and Dalvik. https://source.android.com/devices/tech/dalvik/.

[14] Background optimizations. https://developer.android.com/topic/performance/background-optimization.html.

[15] Battery historian. https://github.com/google/battery-historian.

[16] Cloud9 - automated software testing at scale. http://cloud9.epfl.ch/.

[17] Dextra - A tool for DEX and OAT dumping, decompilation. http://newandroidbook.com/tools/dextra.html.

[18] Distribution dashboards. https://developer.android.com/about/dashboards/.

[19] Es task manager (task killer). https://play.google.com/store/apps/details?id=com.estrongs.android.taskmanager.

[20] Hello daemon. https://github.com/xingda920813/HelloDaemon.

[21] How to turn off smartphone apps that track you in the background. http://www.ibtimes.com/how-turn-smartphone-apps-track-you-background-1657868.

[22] Jeb decompiler by pnf software. https://www.pnfsoftware.com/.

[23] Launch-time performance. https://developer.android.com/topic/performance/launch-time.html.

[24] Matiec - iec 61131-3 compiler. https://bitbucket.org/mjsousa/matiec.

[25] Multicast DNS. https://tools.ietf.org/html/rfc6762.

[26] Nusmv: a new symbolic model checker. http://nusmv.fbk.eu/.

[27] Openplc project. http://www.openplcproject.com/.

[28] Optimizing for doze and app standby. https://developer.android.com/training/monitoring-device-state/doze-standby.html.

[29] Platform Security Architecture. https://source.android.com/security/index.html#android-platform-security-architecture.

[30] Privacy issues: Data abuse on certain mobile apps uncovered. https://www.sciencedaily.com/releases/2012/07/120705133714.htm.

[31] Problem in Making Call Flow Graph from Class or Java files. https://mailman.cs.mcgill.ca/pipermail/soot-list/2014-May/006815.html.

[32] Processes and application life cycle. https://developer.android.com/guide/topics/processes/process-lifecycle.html.

[33] Processes and threads. https://developer.android.com/guide/components/processes-and-threads.html.

[34] ProGuard. http://proguard.sourceforge.net/.

[35] Qualcomm cne to boost snapdragon in carrier wi-fi. https://rethinkresearch.biz/articles/qualcomm-cne-to-boost-snapdragon-in-carrier-wi-fi/.

[36] Ram master – memory optimizer. https://play.google.com/store/apps/details?id=com.speedbooster.optimizer.

[37] Smali An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/.

[38] Smart cooler. https://play.google.com/store/apps/details?id=com.cooler.smartcooler.

[39] Spark – live random chat. https://play.google.com/store/apps/details?id=com.video.chat.spark.

[40] Stack overflow android questions. https://stackoverflow.com/questions/tagged/android.

[41] Super antivirus cleaner & booster. https://play.google.com/store/apps/details?id=com.oneapp.max.

[42] These 5 apps are killing your battery. https://www.androidpit.com/battery-draining-apps.

[43] Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar.

[44] Uppaal home. http://www.uppaal.org/.

[45] Using Network Service Discovery. http://developer.android.com/training/connect-devices-wirelessly/nsd.html.

[46] Vulnerability summary CVE-2006-1856. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-1856.

[47] Xda developers. https://forum.xda-developers.com/android/software.

[48] Xposed development tutorial. https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[49] Industrial control systems killed once and will again, experts warn. https://www.wired.com/2008/04/industrial-cont/, 2008.

[50] Rogue robot blamed for gruesome death of human factory worker it 'trapped and crushed in freak safety failure'. https://www.mirror.co.uk/news/world-news/rogue-robot-blamed-gruesome-death-10026757, 2017.

[51] Mtconnect. https://en.wikipedia.org/wiki/MTConnect, 2018.

[52] Open platform communications. https://en.wikipedia.org/wiki/Open_Platform_Communications, 2018.

[53] Industries. https://www.rockwellautomation.com/en_NA/industries/overview.page, 2019.

[54] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proc. of ACM CCS*, 2015.

[55] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in relay ladder logic programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[56] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of ACM PLDI*, 2014.

[57] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. of ACM CCS*, 2012.

[58] R. Automation. Logix5000 controllers iec 61131-3 compliance. *Rockwell Automation Publication*, 1756.

[59] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of ACM POPL*, 2002.

[60] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proc. of ACM CCS*, 2010.

[61] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *USENIX security symposium*, pages 187–198, 2009.

[62] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl. Regression Verification for Programmable Logic Controller Software. In *Formal Methods and Software Engineering*, 2015.

[63] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.plc: A verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, 2012.

[64] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proc. of ISOC NDSS*, 2012.

[65] E. Byres. The air gap: Scada's enduring security myth. *Communications of the ACM*, 56(8):29–31, 2013.

[66] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of ACM CCS*, 2007.

[67] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of plc programs written in instruction list. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 02 2000.

[68] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of ISOC NDSS*, 2015.

[69] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. USENIX ATC*, 2010.

[70] D. U. Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 2016.

[71] A. Chattopadhyay and K.-Y. Lam. Autonomous vehicle: Security by design. *arXiv preprint arXiv:1810.00545*, 2018.

[72] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of ACM CCS*, 2002.

[73] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proc. of USENIX Security*, 2014.

[74] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *Proc. of the ACM SIGMETRICS*, 2015.

[75] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proc. of the ACM MobiCom*, 2015.

[76] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. of ACM MobiSys*, 2011.

[77] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of USENIX Security*, 2007.

[78] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2010.

[79] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[80] W. Diao, X. Liu, Z. Zhou, and K. Zhang. Your voice assistant is mine: How to abuse speakers to steal information and control your phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 63–74. ACM, 2014.

[81] J. Dzinic and C. Yao. Simulation-based Verification of PLC Programs Master of Science Thesis in Production Engineering. Master's thesis, Chalmers University of Technology, Sweden, 2013.

[82] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proc. of ACM CCS*, 2002.

[83] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones. In *Proc. of USENIX OSDI*, 2010.

[84] S. Etigowni et al. Cpac: securing critical infrastructure with cyber-physical access control. In *ACSAC*, pages 139–152. ACM, 2016.

[85] N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[86] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the ui feedback loop. In *Proc. of the IEEE S&P*, 2017.

[87] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *Proc. of ACM CCS*, 2005.

[88] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proc. of ACM CCS*, 2003.

[89] L. Garcia, S. Zonouz, D. Wei, and L. P. de Aguiar. Detecting plc control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*, Aug 2016.

[90] C. Gibler, J. Crussell, J. Erickson, and H. Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[91] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *Proc. of ISOC NDSS*, 2015.

[92] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. of ISOC NDSS*, 2012.

[93] J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn. The safety guaranteeing system at station hoorn-kersenboogerd. In *Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, Jun 1995.

[94] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of ACM OOPSLA*, 1997.

[95] S. Guo, M. Wu, and C. Wang. Symbolic Execution of Programmable Logic Controller Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, Sep 2017.

[96] N. Hardy. The Confused Deputy:(or why capabilities might have been invented). *ACM SIGOPS*, 1988.

[97] L. Hendren. Scaling Java points-to analysis using Spark. In *Proc. of Compiler Construction, 12th International Conference, volume 2622 of LNCS*, 2003.

[98] R. Huuck. Semantics and analysis of instruction list programs. *Electronic Notes in Theoretical Computer Science*, 115:3–18, 2005.

[99] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy*, 2012.

[100] H. Janicke, A. Nicholson, S. Webber, and A. Cau. Runtime-monitoring for industrial control systems. *Electronics*, 4(4):995–1017, dec 2015.

[101] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity. Contexiot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, 2017.

[102] H. S. Kang, J. Y. Lee, S. Choi, H. Kim, J. H. Park, J. Y. Son, B. H. Kim, and S. Do Noh. Smart manufacturing: Past research, present findings, and future directions. *International Journal of Precision Engineering and Manufacturing-Green Technology*, 3(1):111–128, 2016.

[103] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[104] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[105] R. M. Lee, M. J. Assante, and T. Conway. German steel mill cyber attack. *Industrial Control Systems*, 30:62, 2014.

[106] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction*, pages 47–64. Springer, 2006.

[107] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *Proc. of ISOC NDSS*, 2014.

[108] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of ISOC NDSS*, 2008.

[109] P. Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the USENIX ATC*, 2001.

[110] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of ACM CCS*, 2012.

[111] S. Mani, A. Sankaran, and R. Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 171–179. ACM, 2019.

[112] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *USENIX Annual Technical Conference*, 2015.

[113] S. McLaughlin and P. McDaniel. Sabot: specification-based payload generation for programmable logic controllers. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 439–449. ACM, 2012.

[114] S. McLaughlin, S. Zonouz, D. Pohly, and P. McDaniel. umia. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*, Feb 2014.

[115] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, 2014.

[116] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *Proc. of ACM CCS*, 2012.

[117] J. Nellen, E. Ábrahám, and B. Wolters. A cegar tool for the reachability analysis of plc-controlled plants using hybrid automata. In *Formalisms for Reuse and Systems Integration*, 2015.

[118] J. Nellen, K. Driessen, M. Neuhäußer, E. Ábrahám, and B. Wolters. Two cegar-based approaches for the safety verification of plc-controlled plants. *Information Systems Frontiers*, 18(5):927–952, 2016.

[119] Nellen, Johanna and Driessen, Kai and Neuhäuβer, Martin and Ábrahám, Erika and Wolters, Benedikt. Two cegar-based approaches for the safety verification of plc-controlled plants. *Information Systems Frontiers*, 18(5):927–952, Oct. 2016.

[120] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[121] S. Ould Biha. A formal semantics of plc programs in coq. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*, COMPSAC '11, 2011.

[122] S. C. Park, C. M. Park, G.-N. Wang, J. Kwak, and S. Yeo. Plcstudio: Simulation based plc code verification. *2008 Winter Simulation Conference*, pages 222–228, 2008.

[123] T. Park and P. I. Barton. Formal verification of sequence controllers. *Computers & Chemical Engineering*, 23(11):1783–1793, 2000.

[124] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of the ACM MobiSys*, 2012.

[125] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger. Static code analysis of iec 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application. *IEEE Transactions on Industrial Informatics*, 13(1):37–47, 2017.

[126] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Opportunities and challenges of static code analysis of iec 61131-3 programs. In *ETFA*, pages 1–8, 2012.

[127] J.-M. Roussel and B. Denis. Safety properties verification of ladder diagram programs. *Journal Européen des Systèmes Automatisés (JESA)*, 36(7):pp. 905–917, June 2002.

[128] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[129] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The misuse of android unix domain sockets and security implications. In *Proc. of the ACM CCS*, 2016.

[130] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. IEEE/ACM MICRO*, 2009.

[131] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proc. of IEEE Symposium on Security and Privacy*, 2010.

[132] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139, 2001.

[133] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proc. of USENIX Security*, 2008.

[134] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 832–837, 2018.

[135] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, And Tools (2nd Edition)*. Addison Wesley, 2006.

[136] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[137] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proc. of USENIX HotPower*, 2012.

[138] T. Vennon. Android malware. A study of known and potential malware threats. *SMobile Global Threat Centre*, 2010.

[139] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *Proc. of ACM SIGMETRICS*, 2014.

[140] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current android malware. In *In Proc. of DIMVA*, 2017.

[141] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proc. of the ACM CCS*, 2014.

[142] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proc. of USENIX Security*, 2015.

[143] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proc. of ACM CCS*, 2013.

[144] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proc. of ACM WiSec*, 2012.

[145] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proc. of ACM CCS*, 2015.

[146] L. Zhang, B. Tiwana, R. Dick, and Z. M. Mao. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of ACM CODES+ISSS*, 2010.

[147] K. Zhou, T. Liu, and L. Zhou. Industry 4.0: Towards future industrial opportunities and challenges. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2015 12th International Conference on*, pages 2147–2152. IEEE, 2015.

[148] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proc. of IEEE S&P*, 2014.