Performance, Security, and Safety Requirements Testing for Smart Systems Through Systematic Software Analysis

by

Ke Hong

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

Ke Hong

kehong@umich.edu

ORCID iD: 0000-0002-8830-373X

To Xiaqing and my parents.

ACKNOWLEDGEMENTS

The past five years would be much harder without the guidance from my advisor Professor Zhuoqing Morley Mao, the assistance from my collaborators and colleagues and the support from my family and friends.

Foremost, I would like to express my deepest gratitude to my advisor, Professor Zhuoqing Morley Mao for her continuous kind support and patient guidance on my research. Her constant support was a definite factor in bringing this dissertation to its completion. I still remember she tried different ways to teach me in details how to define meaningful research problems, effectively communicate my ideas in research discussions and conduct impactful research in my early PhD years. Whenever I got lost or stuck in my research, she would always try her best to gear me to the right track and point me to the right direction. Her candid and insightful feedback has been helping me overcome my weaknesses in the research. With her constant guidance and support over these years, I have grown to be a researcher that can independently conduct research.

I would like to thank my other committee members, Professor Scott Mahlke, Professor Florian Schaub and Professor Qi Alfred Chen for their insightful guidance, suggestions and support. They raise important questions on several aspects in this dissertation and provided helpful thoughts to identify a unified theme for this dissertation. With their effort, this dissertation becomes complete and thorough.

I am grateful to be mentored by a few talented industry researchers during my PhD study. It's a pleasure working with Yadi Ma and Sujata Banerjee during my internship at HP Labs in the summer of 2015. I appreciate their help and support for getting my internship work published. I am also appreciative of Mahesh Ketkar and Michael Kishinevsky, for their support and guidance on my prelim project during my internship at Intel in the summer of 2016 and their constant help and feedback to get my internship work published. I also feel thankful to collaborate with a number of academic researchers Professor Scott Mahlke, Professor Harsha V. Madhyastha, Professor Jason Flinn, Professor Peter Chen, Yiheng Feng at Michigan and Scott Haseley, Geoffrey Challen at UIUC. I learned many valuable research and teamwork skills from these mentors and received support in mentoring. I am thankful to PhoneLab's data support of my research.

I am also grateful to my colleagues, especially those who mentored or collaborated in my research projects, and friends at Michigan. It is my honor to work with Mark Gordon during my first semester, who always made himself available to teach me background knowledge of Android, when I had a hard time getting onboard with my research on Android. Ashkan Nikravesh kept encouraging me and providing me feedback to improve the PerfProbe work that eventually led to a publication, even after his graduation. Qi Alfred Chen has been always kind in sharing his experience in system and security research, patiently discussing with me the technical bits and offering useful suggestions to help me make progress for my projects. John Kloosterman and Ze Zhang brought their great expertise in compilers and programing language to my projects and I learnt significant insights from them to improve my research work. I thank Yulong Cao, Shengtuo Hu, Jiwon Joung, Brandon Nguyen, Yuqi Jin for their help on my projects in autonomous vehicles and Yunhan Jia, Yuru Shao, Yan Chang, Shaobing Xu for their knowledge sharing of systems security, Android platform and autonomous vehicle systems. I also thank Sanae Rosen, Yihua Guo, Mehrdad Moradi, Hongyi Yao, Shichang Xu, Yikai Lin, Chao Kong, Xiao Zhu, Jie You, Jiachen Sun, Won Park, Can Carlak, Xumiao Zhang, for their constant support and help with my research and life at Michigan. In addition, I appreciate for the company and support offered by my friends I made in my PhD study and internship, Sheng Liu, Chengxue Shao, Da Yu, Zhiqiang Sui and Yu-Chih Tung.

I am thankful to my master thesis advisor Professor Lin Gu, former research colleague Zhiqiang Ma, undergraduate thesis advisor Professor Qian Zhang at Hong Kong University of Science and Technology. They patiently taught me in details how to be a system hacker and researcher in my early days of research study.

Finally, I would like to express my deepest thanks to my girlfriend Xiaqing Jiang and my father, Hairong Hong, mother, Shunyu Shen for being the constant pillars of support in my life. Xiaqing has been always by my side with great understanding and support for me to accomplish my goals during my doctoral study. My parents gave constant support and understanding to let me focus on the pursuit of my graduate studies. Their unreserved love and support helped me bring this adventure to its end. This dissertation is dedicated to them.

I would also like to acknowledge NSF grants and the Naval Research grants for their support of my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Smartphones, wearable devices and emerging autonomous vehicles (AVs) are significantly transforming our way of communication, networking, knowledge acquisition, healthcare and transportation. As our daily lives are increasingly relying on these *smart end systems*, certain guarantees on the performance, security and safety becomes critical requirements to the design and implementation of the software for these systems. To ensure such key requirements are met before shipping the software into users' devices/vehicles, it is necessary to exhaustively test and verify the software at the development and testing stage. However, testing and verifying the performance, security and safety requirements for the software of these systems remains a research challenge. Due to the high mobility of these systems in the real world, the runtime environments faced by these systems vary significantly, which poses challenges to the testing and validation of performance requirements. Also, due to the layering design fashion and multi-party development process, software running on these systems is usually highly complex, potentially enlarging attack surface and posing challenges to the testing and validation of security and safety requirements.

To address this challenge, this dissertation focuses on developing systematic and automated software analysis tools for testing the performance, security and safety requirements of the software for smart end systems. Specifically, we demonstrate that automated program analyses based on 1) static program analysis and 2) runtime program profiling with certain system domain-specific customization, can lead to effective testing and validation of key performance, security and safety requirements for smart system software.

This dissertation contributes to the performance, security and safety requirements testing of smart end systems in following aspects: (1) effectively test performance requirements and diagnose the cause of performance slowdown through lightweight monitoring of and systematic performance characterization based on cross-layer runtime events, (2) systematically detect noncompliance with important security principles (e.g., publish-subscribe overprivilege vulnerability) through systematic program analysis and mitigate security vulnerabilities through policy enforcement, and (3) systematically verify the compliance with safety requirements on the mission-critical components (e.g., AV's driving decision control) of smart end systems.

CHAPTER I

Introduction

Smart end systems operate in the wild, directly interact with users and are equipped with increasing processing power, connectivity and intelligence. Real-world systems of this category include smartphones, wearable devices and emerging autonomous vehicles (AVs) and are reshaping our daily lives tremendously. The last decade has seen how smartphones, smartwatches and smart home devices revolutionize the way we acquire information and communicate with each other. Nowadays various mobile devices have become ubiquitous and the number of mobile apps in app markets keep multiplying. Today's Google Play store stocks over 2 million Android apps with over 50 billions of total downloads [65, 41]. Moreover, the recent emergence of AVs holds great promise in transforming current transportation systems for better road safety and mobility efficiency and enabling future mobility services. Recently, highly autonomous driving systems are being deployed in real vehicles [106, 78, 97, 15] and have demonstrated potential of mass production in the coming decade.

Due to its increasing ubiquity and deployment for mission-critical purposes, performance, security and safety become critical requirements for the design and implementation of smart end systems. First, performance is an important requirement for achieving desired interactivity of these systems, since they either are user-facing or make mission-critical decisions in response to the change of physical environment in low-latency control loops. Good quality-of-experience (QoE) for mobile apps (e.g., has been shown crucial for achieving good user experience and improve app engagement. Google's RAIL performance model [98] indicates that a user may lose focus on the task they are performing if the system response takes over 1 second. In order to achieve fast reaction to the change of physical surroundings in self-driving, software modules of an AV system needs to go through a control loop including sensor input acquisition and processing, machine learning based object prediction, optimization-based path planning and control decision actuation in as tight as 10 to 100 milliseconds. Such low latency requirements pose stringent challenges to even today's commodity operating systems and device hardware, not to mention that they need to be placed within a space-constrained platform like a smartphone or vehicle.

Second, these systems face security and privacy threats given their high software complexity for supporting rich functionalities and the increasing number of communication and sensing interfaces exposed on them. New programming defects or attack surface can occur due to the increasing software complexity. A broad range of security vulnerability and privacy leakage, due to the open development nature, the coarse granularity of permission control, etc., have been uncovered in smartphone systems. Automotive systems also face a number of cybersecurity threats. Security vulnerabilities on automotive Controller Area Network (CAN bus) and Electronic Control Units (ECU) have been uncovered that lead to remote compromise and control of a wide range of automotive functions and completely ignore driver input [162, 125, 144, 130, 44, 36, 68, 63]. Moreover, the various sensors equipped on AVs can expose new attack surface with severe security implications [174, 189, 204, 188, 178, 91]. Besides these known attack surfaces, another highly critical yet less explored AV-specific attack surface, the AV software systems for making autonomous driving decisions, can be potentially exploited when more and more advanced autonomous driving algorithms and functionalities incorporated into its code base.

Third, unanimously agreed by AV vendors and transportation authorities, strong safety guarantees for the self-driving control are at the core of the design and implementation of AV software systems. Nowadays, AV software is the main component responsible for making real-time mission-critical driving control decisions for an AV to avoid crash and comply with safety driving practices. Specifically, certain safety policies, such as NHTSA's safety elements for AVs [17], existing traffic laws, etc., need to be correctly enforced in the key self-driving modules of an AV software system. Since these software systems generate physical driving actions that have direct impact on road safety, it is necessary to understand potential security vulnerabilities and safety policy violation in the design and implementations of AV software systems, and proactively address them in the AV system development stage.

Besides the software complexity, some smart platforms, e.g., Android, Baidu Apollo, etc., encourage open-source contribution from third-party developers, making the compliance with these key requirements difficult at development stage. In the AV context, things get worse because AV software development is a multidisciplinary task and typically conducted by a large team of developers with different domains of expertise. Therefore, before the real-world deployment, the compliance with performance and safety requirements and key security properties needs to be rigorously verified in the lab. Moreover, performance requirements testing and validation needs to be extended to the stage of deployment given that it is highly correlated to the user runtime, where network or server conditions may vary drastically and physical resource constraints on individual devices/vehicles may differ.

However, testing and validating performance, security and safety requirements for these smart systems remains a research challenge due to their unique characteristics. First, individual devices/vehicles spread in the wild and are highly mobile at run time. As a result, testing the performance requirements and diagnosing the root cause for any noncompliance requires holistic yet lightweight performance monitoring and policy enforcement at run time. Effective diagnosis should lead to useful and actionable findings that can help app and AV developers localize the executed code causing performance degradation and also understand the cause of execution slowdown from a system perspective, e.g., due to variable network or server factors or device-specific resource bottlenecks (e.g., CPU, memory, I/O, etc.). Second, the software stack of these systems, such as the AOSP framework [2] for smartphones, Baidu Apollo framework [14] for AVs, etc., commonly adopts the layering design fashion to reduce the overall system complexity and improves resource management and programability. Besides, third-party libraries are extensively used and increases the code complexity of the software stack. Both the layering design and high code complexity pose new challenges to the systematic testing of certain requirements and detection of any noncompliance for the smart systems. One challenge is the completeness guarantee in static code analysis for achieve systematic analysis goals. Another is the need of monitoring runtime events across multiple layers of a system for program profiling to gain holistic root cause understanding for performance issues. For example, mobile apps at the run time may frequently interact with the underlying operating system to access certain types of resource and their execution performance can be constrained by the shortage of a desired system resource. To better guide the root cause analysis of performance requirement noncompliance, runtime profiling should capture not only app-level execution traces but also system-wide events for localizing easy-to-reason code-level bottleneck and pinpointing the system-level resource bottleneck. Third, some important security properties and safety requirements, e.g., traffic rules in human language specification, are too abstract to be easily defined using a code-level representation that can be directly verified in the software stack of these systems. Domain-specific abstraction and mapping are required to bridge the semantic gap between the specification of safety/security properties and code-level implementation, such that a requirement can be formulated as specification in code-level constructs and verified in the software implementation of smart systems.

My dissertation is to address this challenge by developing systematic and automated software analysis support for smart end systems. Specifically, my approach leverages two main types of analysis techniques, 1) static program analysis to achieve completeness guarantees of analyzing program behaviors and 2) runtime program profiling to capture runtime conditions of program execution. By applying both techniques, my dissertation demonstrates that **systematic software analysis approaches based on static program analysis and runtime profiling techniques, with certain domain-specific customization, can lead to effective testing of key performance, security and safety requirements for smart system software**: (1) effectively test performance requirements and diagnose the cause of performance slowdown through lightweight monitoring of and systematic performance characterization based on cross-layer runtime events, (2) systematically detect non-compliance with important security principles (e.g., publish-subscribe overprivilege vulnerability) through systematic program analysis and mitigate security vulnerabilities through policy enforcement, and (3) systematically verify the compliance with safety requirements on the mission-critical components (e.g., AV's driving decision control) of smart end systems. Table 1.1 summarizes the key problem and analysis technique used in each project demonstration.

Scope	Problem & Requirement	Analysis technique
Smartphone	Unpredictable slowdown diagnosis (**performance**)	Runtime program profiling
Autonomous vehicle	Publish-subscribe overprivilege analysis (**security**)	Static program analysis
Autonomous vehicle	Safety driving rule compliance verification (**safety**)	Static program analysis

Table 1.1: Overview of smart systems covered in this dissertation.

This dissertation proposes a unified software analysis framework (illustrated in Figure 1.1) for smart system software vendors and developers to achieve systematic testing of performance, security and safety requirements of smart system software. This framework is customizable to incorporate smart system domain-specific knowledge to improve the effectiveness of requirement testing based on runtime profiling and static program analysis techniques. Our project demonstrations incorporate a set of problem and system domain-specific customization to both analysis techniques to addresses following challenges with testing the performance, security and safety requirements of certain smart systems.

- For performance requirement testing with smartphone systems, we propose a new adaptive sampling technique to limit the performance overhead within 3.5% increase

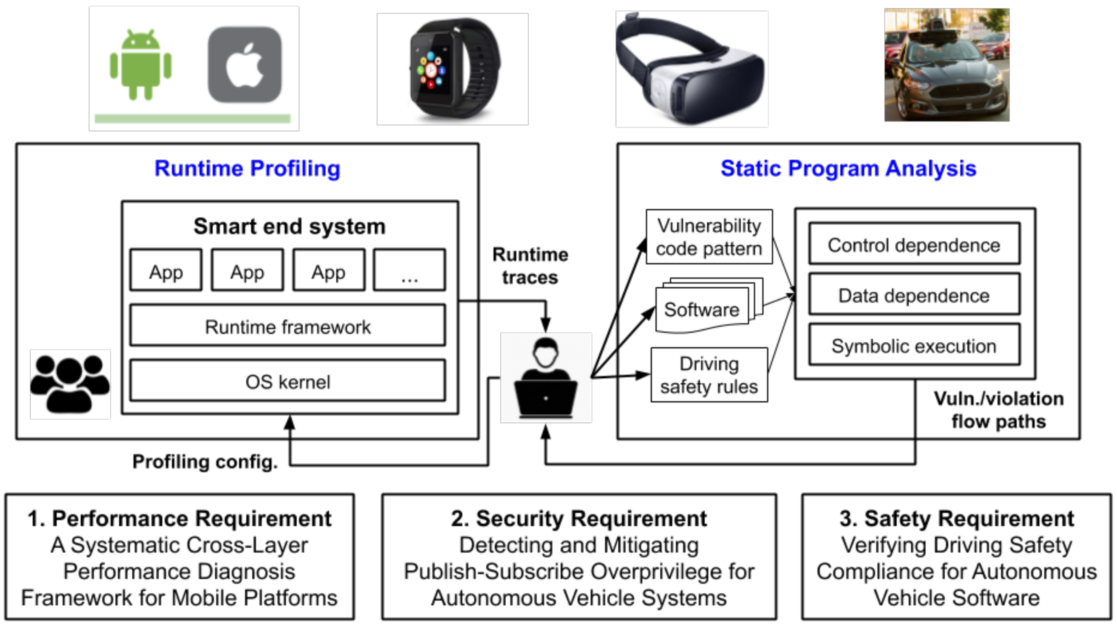

Runtime Profiling
Smart end system
App
App
App
...
Runtime framework
OS kernel
Runtime traces
Profiling config.
Static Program Analysis
Vulnerability code pattern
Software
Driving safety rules
Control dependence
Data dependence
Symbolic execution
Vuln./violation flow paths
1. Performance Requirement
A Systematic Cross-Layer Performance Diagnosis Framework for Mobile Platforms
2. Security Requirement
Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems
3. Safety Requirement
Verifying Driving Safety Compliance for Autonomous Vehicle Software

Figure 1.1: Software analysis framework to support systematic testing of performance, security and safety requirements for smart end systems.

of user-perceived delays for user interactions of 100 popular Android apps.

- For security vulnerability analysis of AV software systems, we address analysis challenges caused by the broad use of object-oriented programing principles and event-triggered asynchronous model in AV software to achieve *zero* false negative in detection of an overprivilege vulnerability.

- For driving safety compliance testing, we incorporate AV-specific domain knowledge (a.k.a., traffic scenarios and driving actions) to allow flexible specification of driving safety rules, as target code-level patterns.

The contributions of this dissertation are elaborated in the following three sections.

1.1 *Addressing Performance Requirement*: Diagnosing Unpredictable Performance Slowdown with Mobile Apps

User-perceived performance slowdown in mobile apps can occur in unpredictable and sophisticated ways, with root cause spanning at different layers (app or system layer). There is a lack of effective approaches to provide cross-layer, holistic insights to diagnose **unpredictable performance slowdown** on mobile platforms, motivating us to develop PerfProbe as a performance diagnosis framework for mobile platforms. PerfProbe monitors app performance and records app and system-layer runtime information in a lightweight manner on mobile devices, and performs systematic, novel statistical analysis on collected runtime traces at different layers to localize code-level performance variance in the form of *critical functions* and zoom into them to pinpoint system-level root causes in the form of *relevant resource factors* to explain the performance slowdown. PerfProbe effectively diagnoses performance slowdown due to various root causes in 22 popular Android apps from real-world usage monitoring and in-lab testing, by providing holistic, cross-layer insights to help the root cause diagnosis. Diagnosis findings from PerfProbe provide actionable insights for root cause finding and guiding real-world app developers' code fixing or adjustment of platform-level policies to reduce user-perceived latency of 6 real Android apps by 32-86%. PerfProbe incurs small system overhead and impact to app performance at runtime and is suitable for real-world deployment.

1.2 *Addressing Security Requirement*: Overprivilege Analysis and Policy Enforcement for Autonomous Vehicle Systems

Autonomous vehicle (AV) software systems are emerging to enable rapidly developed self-driving functionalities. Since such systems are responsible for safety-critical decisions, it is necessary to secure them in face of cyber attacks. Through an empirical study of representative AV software systems *Baidu Apollo* and *Autoware*, we discover a common *overprivilege* problem with the publish-subscribe communication model widely adopted by AV systems: due to the coarse-grained message design for the publish-subscribe communication, some message fields are over-granted with publish/subscribe permissions. To comply with the least-privilege principle and reduce the attack surface resulting from such problem, we argue that the publish/subscribe permissions should be defined and enforced at the granularity of *message fields* instead of messages.

To systematically address such publish-subscribe overprivilege problem, we present AVGuardian, a system that includes (1) a static analysis tool that detects overprivilege instances in AV software and generates the corresponding access control policies at the message field granularity, and (2) a low-overhead, module-transparent, run-time publish/-subscribe permission policy enforcement mechanism to perform online policy violation detection and prevention. Using our detection tool, we are able to automatically detect 579 overprivilege instances in total in Baidu Apollo. To demonstrate the severity, we further constructed several concrete exploits that can lead to vehicle collision and identity theft for AV owners, which have been reported to Baidu Apollo and confirmed as valid. For defense, we prototype and evaluate the policy enforcement mechanism, and find that it has very low overhead and does not affect original AV decision logic.

1.3 *Addressing Safety Requirement*: Safety Rule Verification for Autonomous Vehicle Software

As we are getting to a feasible stage of running our daily transportation and mobility services using AVs on real-world roads, safety requirement is paramount in the design and implementation of AV software systems. A set of key safety elements [17] defined by the National Highway Traffic Safety Administration (NHTSA), including the safety rules in existing traffic laws, forms the voluntary safety standards for autonomous driving systems to comply with. However, in reality, compliance with these safety policies is hard to be enforced an verified at the software level. One main reason is the increasing complexity of AV software due to new features to be supported and the multidisciplinary development nature in AV software. The open ecosystem promoted in certain AV developer communities (e.g., Baidu Apollo [14]) also makes the policy compliance difficult to guarantee.

To help AV developers verify the compliance with predefined safety requirements of their software before real-world testing, we develop AVerifier, a static program analysis framework towards verifying the compliance of user-defined safety requirement in AV software with completeness guarantees (e.g., full code coverage). Specifically, we design and implement a program dependence analysis framework to help AV developers perform static detection of safety requirement noncompliance problems in AV software. Our analysis uncovers the noncompliance with some well-recognized safety requirements (e.g., compliance with traffic laws) in an early version of Baidu Apollo. To allow expressive specification of safety requirements, a specification interface is proposed based on a unique abstraction of the rich road traffic and driving semantics and a semantic mapping that relates each semantic entity to its code-level implementation in AV software. We validate that this interface enables AV developers to express a broad range of common safety rules from existing traffic regulation documents in a composable and flexible manner.

1.4 Approach Generality

The customized runtime profiling and static program analysis techniques are in principle applicable to both smart end systems studied in our project demonstrations (i.e., smartphone and autonomous vehicle). As presented in Chapter II, both smartphones and AVs share the layering design fashion in their software stacks. Though §1.1 demonstrates the performance requirement testing and problem diagnosis in the context of Android smartphone systems, the demonstrated runtime profiling and diagnosis techniques are in principle applicable to the performance requirement testing of AV software systems. However, the location of instrumentation need to be specific to the implementation of an AV system (e.g., Baidu Apollo's runtime framework and customized Linux kernel [14]) to achieve effective our cross-layer runtime profiling. Moreover, due to the common use of object-oriented programing principles in both smartphone and AV software systems, the customized static program analysis techniques presented in §1.2 and §1.3 by design can be generalized to smartphone systems for detecting similar security vulnerabilities (e.g., publish-subscribe overprivileged if exists) and verifying security policies (e.g., access control policies). Yet, as highlighted by the thesis statement of this dissertation, smartphone-specific domain knowledge should be considered to achieve systematic analysis goals of the static detection in the smartphone software context.

1.5 Lessons Learnt

Based on our research investigation on the three projects, we summarize our general takeaway on the advantages and limitations of both analysis techniques on the requirement testing of smart end systems.

- Runtime profiling, as demonstrated by PerfProbe, is able to capture the dynamic runtime environments in a holistic manner even under the high mobility of smart end systems (e.g., smartphones). It is thus well suited for testing and analyzing performance properties significantly influenced by the system runtime. However, performance overhead remains a key barrier for their deployability in smart end systems. From our empirical study, we discovered that Android's built-in app profiling tool [72] may introduce up to 22% increase of user-perceived latency for some user interactions in commercial Android apps. Also, as profiling certain layers (e.g., call stack) may require high system privilege, e.g., a rooted smartphone, deployability issues and usage scenarios should be a primary concern when incorporating runtime profiling techniques for software analysis.

- Static program analysis by design is capable of providing completeness guarantee in analyzing possible program behaviors. Given this advantage, it is particularly useful for testing security properties to find vulnerable flows and paths of software code or verifying compliance with predefined rules in the code-level implementation in general software systems. However, one prerequisite to perform static program analysis in target software is a precise definition of target code-level patterns, which is usually specific to an analysis goal and a target software system. Defining the patters may require some levels of domain knowledge. For example, *AVerifier* requires common traffic and driving domains for building driving safety rule abstractions and non-trivial implementation domain knowledge to construct the semantic mapping for code-level rule specification. Moreover, static program analysis requires definition of entry points (a.k.a., sources) and sinks for the code analysis to start with and terminate on. Identifying sources and sinks of software for new smart systems may be non-trivial and requires prior knowledge specific to the system domain. Based on our research investigation, the life cycle of an app/module and their used APIs for performing relevant actions (e.g., publish-subscribe, self-driving control actions), can be generalized as a common source-sink pattern across different system implementations and domains.

- Due to the use of advanced programming paradigms (e.g., object-oriented programming principles, event-triggered asynchronous model) in smart systems and lack of runtime conditions at compile time, conservative assumptions on the flows of a program becomes unavoidable for static program analysis to achieve full code coverage and complete approximation of program behaviors, which however may lead to over-approximation of the code-level pattern detection. In the context of security vulnerability or safety violation detection, over-approximation usually results in high false positive rate and may thus affect the tool usability from the developer perspective.

1.6 Thesis Organization

This dissertation is structured as follows. Chapter II provides sufficient background of Android platform, AV software architecture and key program analysis techniques used across this dissertation. In Chapter III, we describe a lightweight profiling approach for testing performance requirement in real-world usage and systematic characterization approach for diagnosing root cause for noncompliance. In Chapter IV, we conduct our first systematic analysis of publish-subscribe overprivilege in popular open-source AV software systems. In Chapter V, we present our a program analysis approach to verify safety compliance in AV software. We discuss related work in Chapter VI before concluding the thesis in Chapter VII.

CHAPTER II

Background

2.1 Android Platform

This section provides sufficient background of mobile operating systems and Android's framework. Android is a mobile operating system developed by Google [2]. It is based on a modified version of the Linux kernel and other open source software, and is designed primarily for smartphones and tablets, but recently extends the support to televisions (Android TV), cars (Android Auto), and smartwatches (Android Wear OS). Its software stack follows the layering design illustrated in Figure 2.1. From bottom to top, it consists of 4 software layers.

- **Linux kernel**. This layer features core operating system (OS) services and device drivers that allow the OS to access the low-level hardware.
- **Android runtime and native libraries**. Android runtime features a Dalvik virtual machine (DVM) that executes the Dalvik bytecode of Android apps, similar to Java virtual machine (JVM) for executing Java bytecode. Native libraries are shipped with native code that can be executed by the CPU directly to achieve high runtime efficiency.
- **Application framework**. This layer provides an interface for Android apps to access the system resources. It consists of a number of key services, including ActivityMan-

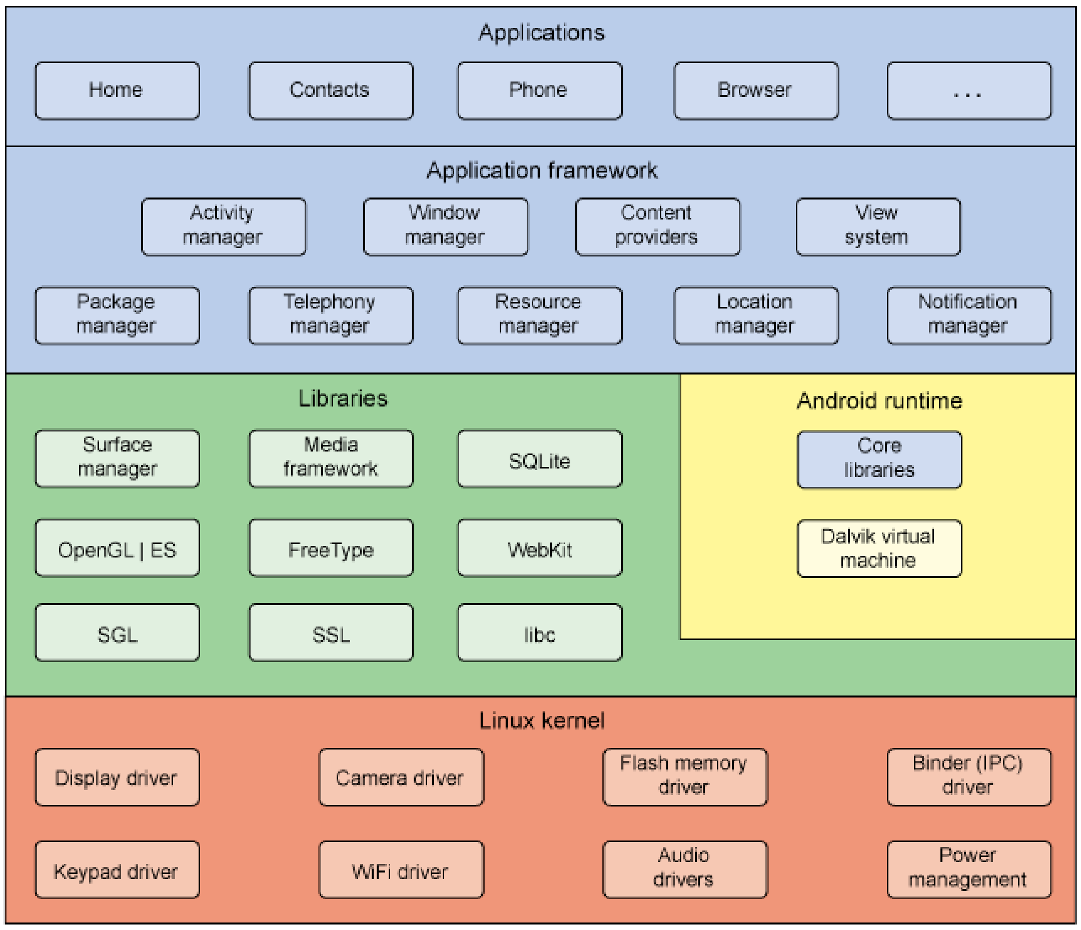


Figure 2.1: Overview of Android architecture [3]

ager to manage the life cycle of Android apps, LocationManager to support Android apps retrieving GPS coordinates of a device, etc. Android apps can access these services through the Android SDK.

- **Applications**. This layer supports all the apps that users interact with, including system apps shipped by device vendors and third-party apps from Android app markets (e.g., Google Play).

This dissertation demonstrates performance requirement testing and diagnosis of Android apps, by monitoring and analyzing app and system-level runtime events when a user interaction is performed and executed in an Android app. Specifically, we design a lightweight monitoring framework to perform runtime profiling across the four layers of the Android

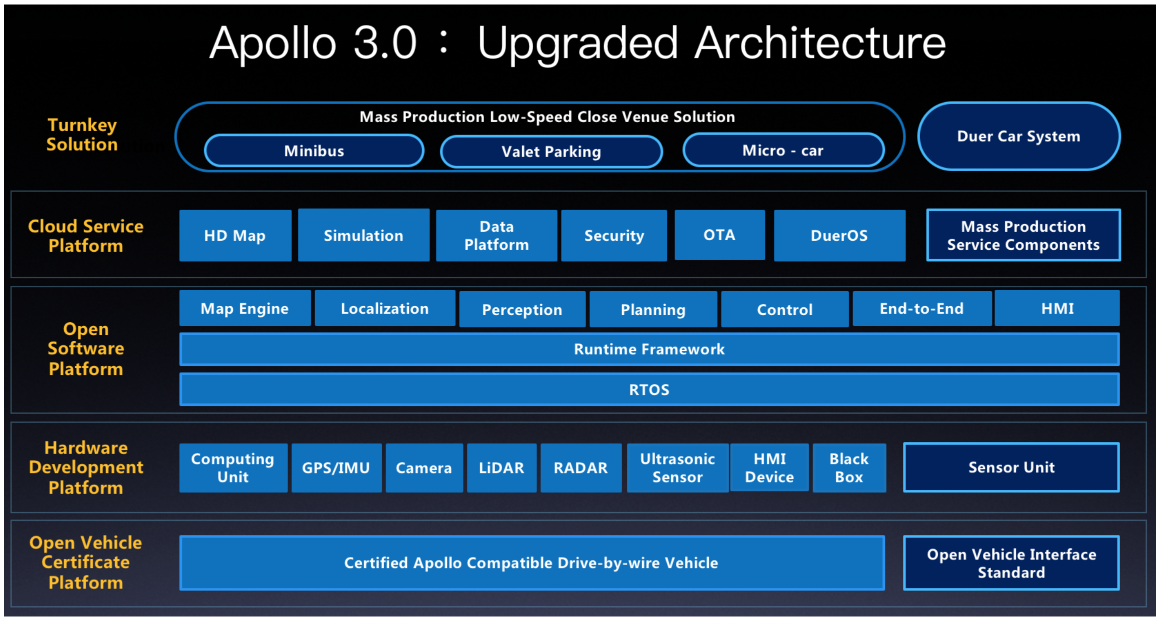


Figure 2.2: Overview of AV architecture in Baidu Apollo v3.0 [14]

stack.

2.2 Autonomous Vehicle Systems

This section provides sufficient background of autonomous vehicle architecture and how the self-driving functionalities are achieved in this architecture. Figure 2.2 shows a full stack of typical software and hardware architecture of current AV systems. This dissertation focuses on the software platform layer in the figure. In this layer, from top to bottom, consists of key software modules implementing the self-driving control pipeline (detailed in §4.2), a runtime framework/middleware for efficient inter-module communication and flexible resource management, and a real-time operating system (RTOS) with a customized kernel.

One popular middleware used in AV systems is Robot Operating System (ROS). We define some common terms of this middleware that are used in the following chapters. Figure 2.3

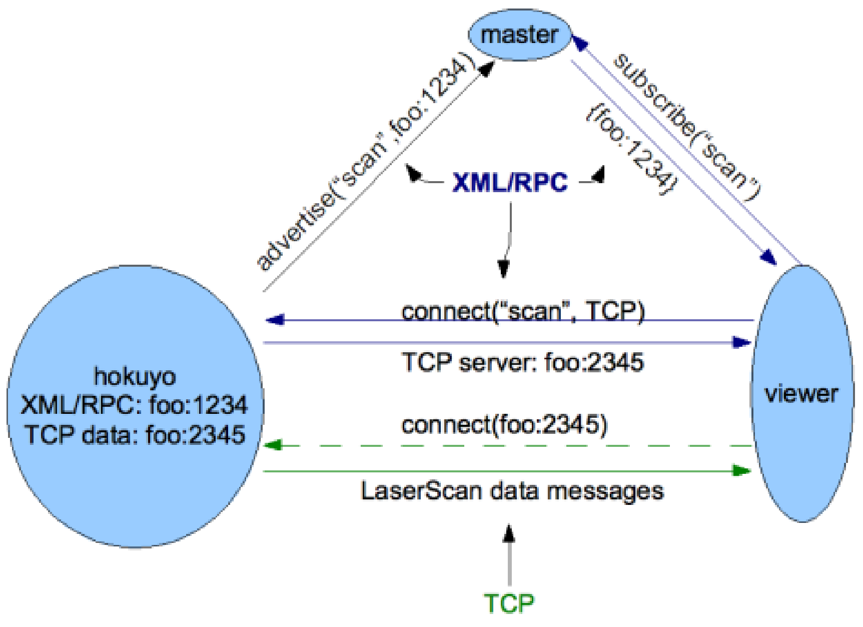


Figure 2.3: Publish-subscribe topic transport in ROS [76]

- **Node**: A node is a process that performs computation and has a graph resource name that uniquely identifies it to the rest of the system. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, or the Parameter Server. Usually, an AV system comprise a number of nodes to control different functions.

- **Message**: Nodes communicate with each other through messages. A message contains data that provides information to other nodes. Besides the standard message types defined in a middleware (e.g., ROS), new types of messages can be defined in the self-driving modules of an AV software system using standard message types.

- **Topic**: Topics are named buses over which nodes exchange messages over a publish-subscribe communication channel. Each message must have a name to be routed by the ROS network. When a node is sending data, we say that the node is publishing a topic. Nodes can receive topics from other nodes by simply subscribing to the topic. Figure 2.3 illustrates the setup of this publish-subscribe communication channel in

ROS. In ROS, all messages on the same topic must be of the same data type.

This dissertation demonstrates security and safety requirement verification on the software design and implementation of the self-driving control pipeline and the middleware of open-source AV systems.

We draw following common observations on smartphone and AV systems and characterize some commonalities of the software for smart end systems.

- A majority of smart end systems, including smartphones, wearable devices and AVs, are deployed in the wild and highly mobile at the run time. Thus, the performance requirement testing for smart systems need to capture a holistic view of runtime environments and dynamics.

- The layering design adopted by smart end systems increases the complexity of software analysis. On the one hand, ideally all layers should be considered for capturing holistic runtime conditions, which however may lead to high performance overhead for the runtime profiling. On the other hand, the layering design makes interprocedure as a necessary analysis requirement for static program analysis of smart systems software and increases the complexity of control and data flow analysis.

- Most smart end systems are developed following object-oriented programing principles (e.g., inheritance and polymorphism) and the event-triggered asynchronous model. As a result, the runtime behaviors need to be conservatively modeled for achieving the completeness for flow-sensitive analysis. Additional program analysis techniques, e.g., symbolic execution, may be required to reduce the over-approximation of static program analysis.

2.3 Program Analysis Techniques

This dissertation leverages two main types of program analysis techniques: 1) static program analysis, and 2) runtime program profiling. We briefly summarize the pros and cons of each technique as follows.

- **Static program analysis** is capable of automatically analyzing the behavior of a program regarding certain program property, e.g., performance [109], correctness [164], and security [128], by examining its source code or binary without executing it. Depending on the analysis goals and requirements, such analysis can be performed statically at the compile time for predicting computable approximations to the targeted property The program behaviors to be analyzed commonly need to be precisely defined as some code-level patterns. On the one hand, it usually holds great potential in checking a target property with full code coverage and has been widely used as a systematic approach for various security problems, including discovering vulnerabilities such as buffer overflow [133] and cross-site scripting [196], detecting privilege escalation attack in access control systems [187], detecting privacy leakage [146, 113], detecting and analyzing malware [190], etc. On the other hand, the static nature makes it tend to over-approximate the behavior of a program and may incur false positives in problem detection [113, 187].

- **Runtime program profiling** is a form of dynamic program analysis to measures certain properties of a program, e.g., the memory consumption of a program, the usage of particular instructions, or the frequency and duration of function calls. The runtime information gathered from profiling can produce insights that are not derivable from static analysis, especially when target properties are highly dependent on specific runtime environments (e.g., network condition, hardware platform, etc.). However, profiling may introduce substantial overhead to the system runtime and impact application performance [72, 34]. Profiling can be achieved by trapping certain events (a.k.a., event-based), instrumenting the program source code or its binary (a.k.a., instrumented), or probing the target program's call stack at regular intervals using operating system interrupts (a.k.a., sampling). Each technique has its advantages and drawbacks in accuracy and overhead.

CHAPTER III

PerfProbe: A Systematic Cross-Layer Performance Diagnosis Framework for Mobile Platforms

3.1 Introduction

With the rapid advancement of mobile computing and networking technologies, mobile devices have become ubiquitous and the app market is multiplying. Today's Google Play store stocks over 2 million Android apps with over 50 billions of total downloads [65, 41]. Different from server-based applications, mobile apps are user-facing and highly interactive, and usually running on a resource-constrained and dynamic environment. These unique runtime features make apps more likely to be affected by internal device-specific resource constraints (e.g., CPU, memory, disk) as well as external environment factors, including network quality and server-side delay. Variance of some key factors may lead to large variation in user-perceived latency, degradation of which is an important type of quality-of-experience (QoE) problems for mobile apps. As one critical performance metric for a wide variety of interactive apps, user-perceived latency has recently drawn attention from the research community [126, 168] and app industry [64]. Google's RAIL performance model [98] shows that a user may lose focus on the task they are performing if the system response takes over 1 second. Therefore, it becomes crucial to uncover performance degradation in critical user interactions and diagnose them at the early testing or deployment stage, so as to provide app developer or device vendors with useful hints for implementing effective strategies to ensure app responsiveness and good user experience.

Our empirical study on 100 popular Android apps shows that user-perceived performance for key user interactions in an app can degrade by multiple times in some runs. As later shown by our diagnosis, such unpredictable performance slowdown can be due to specific runtime context (§3.6.3) or code-level design issues (§3.6.1). Pinpointing these sophisticated factors requires analyzing app and system-layer information collected at runtime. On the one hand, localizing code-level performance variance with common program abstractions (e.g., function) provides semantically meaningful hints for root cause reasoning and code fixing by human developers. On the other hand, system-wide runtime events record fine-grained details on how an app interacts with system resources over time, which may help app developers quantify the runtime cost of their code to certain types of resource, especially when an invoked third-party library is proprietary or too complex to understand its resource intensity. Such resource-level root cause reasoning is also useful to guide device vendors' refinement of system-level configurations or policies (e.g., buffer size, frequency governor, code offloading [134, 148, 147]) to achieve better mobile performance.

Unfortunately, we see a lack of effective approach to provide such cross-layer, holistic insights for helping the diagnosis of unpredictable performance slowdown on mobile platforms. To fill this gap, we develop PerfProbe as a performance diagnosis framework for mobile platforms. PerfProbe does not require app source code and takes app binary as input. Developers can specify their interested user interactions to be monitored through its configuration interface (§3.3.1). PerfProbe then monitors performance of these interactions triggered by real-world usage and records app and system-layer runtime information in a lightweight manner on a mobile device. Once performance slowdown is detected, PerfProbe performs offline diagnosis by associating the collected runtime traces at different layers using a statistical learning approach. PerfProbe provides cross-layer, informative insights as its diagnosis output to facilitate the root cause analysis by app developers or device vendors: 1) *critical function* showing the executed function calls whose slowdown is most correlated to performance degradation; 2) *relevant resource factor* indicating what system resource (e.g., computation, network, disk, etc.) a critical function interacting with is most correlated to the slowdown of the function. Motivated by a real-world app study and subsequent diagnostic evaluation, our cross-layer characterization approach enables resource-level understanding compared to existing app-level profiling approaches [183] (§3.2.1), and achieves higher accuracy in pinpointing the relevant resource factors causing performance slowdown than existing OS monitoring or resource profiling approaches [206, 39, 67, 159] (§3.6.4).

To develop PerfProbe, we overcome two major research challenges. First, recording fine-grained app and OS-layer runtime information can incur large overhead to a mobile device and degrades app performance, influencing both user experience and accuracy of problem diagnosis. To address this challenge, we propose a novel model-driven adaptive sampling mechanism to accommodate the different levels of profiling overhead incurred by different apps on different devices and achieve lightweight call stack profiling. It performs real-time monitoring of the performance impact to an app due to profiling its call stack and based on that adjusting the call stack dumping frequency to limit its impact within some configurable threshold (§3.3.2). Second, a user interaction in real apps usually involves execution across dozens or hundreds of threads and even across process boundary, with different threads bounded by different system resources. To overcome this challenge, we propose a novel statistical analysis approach that first zooms into app-level execution to identify a small set of critical functions and then pinpoints their underlying resource factors relevant to the cause of performance variance (§3.4). As one main novelty of our system, this two-step critical function and resource factor characterization imitates human inspection, but involves no manual efforts.

Our work makes the following research contributions:

- We develop a lightweight performance monitoring mechanism with smaller performance impact than state-of-the-art for mobile platforms that collects detailed app and system-layer runtime information to support the diagnosis of unpredictable performance slowdown in mobile apps.
- We design an automated, systematic cross-layer characterization approach that performs two-step statistical characterization on app and OS-layer traces to pinpoint critical functions and their underlying relevant resource factors for explaining the cause of unpredictable performance slowdown in mobile apps. This cross-layer characterization approach by design can be generalized to diagnosing similar performance issues in other software systems.
- Diagnosis findings from PerfProbe on real-world performance issues provide valuable insights for guiding code-level fixing of real-world app developers and adjustment of platform-level policies to reduce user-perceived latency of 6 real Android apps by 32-86%.

In the following sections, we will use the term performance or user-perceived latency interchangeably.

3.2 Motivation & Approach

In this section, we motivate the need of associating app and OS-layer runtime information for performance analysis using a popular Android app as a motivating example (§3.2.1) and present a key design challenge to achieve our diagnosis goal (§3.2.2).

3.2.1 Motivating Example

We study *SSE*, a popular Android encryption app, in which users click a UI button to encrypt a file stored in SD card. Perturbing different resources in the system consistently causes severe performance degradation. Figure 3.1 illustrates the execution workflow for

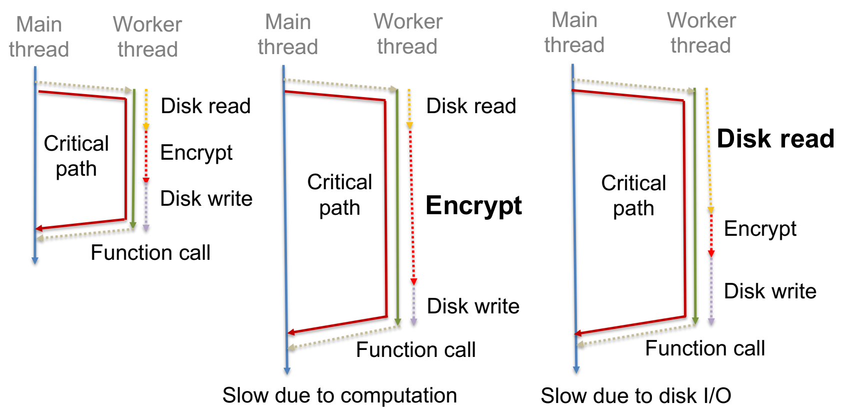


Figure 3.1: Performance slowdown due to different root causes (length of the arrows corresponding to execution time) in Run 1, 2, 3 (from left to right). No slowdown in Run 1.

this interaction in different runs. In this interaction, the main/UI thread invokes a worker thread to execute an encryption function that performs three operations: read the file from SD card, encrypt the bits and write the encrypted file to SD card. Performance slowdown that occurs in 2nd and 3rd run, however, are due to different resource bottlenecks – slow disk I/O in loading the file from the local storage in one run and insufficient CPU cycles for performing computation-intensive encrypting operations in the other run.

The benefits of associating app and OS-layer runtime information together for performance diagnosis are two-fold. First, app-level profiling [72, 183] may identify what function calls lead to performance variance (critical functions SCrypt.scryptN or Posix.readBytes in Table 3.2) under different resource perturbations, but is unable to pinpoint underlying resource bottlenecks that are unique to the runtime. Domain knowledge on the script or posix library is required for understanding. In fact, performance slowdowns observed in our deployment study were caused by app's invocation of certain system resources that become a bottleneck (e.g., computation bottleneck due to the CPU frequency cap enforced by DVFS governor policies in §3.6.3.1, disk I/O bottleneck due to the read-ahead buffer limit in §3.6.3.2, etc.), which can hardly be uncovered by analyzing app-layer execution alone. Second, applying traditional resource profiling [39, 67] or tracking system calls [159] or primitive OS events [206] loses track of details of program semantic (e.g., functions) and cannot provide developers with easy-to-reason hints to enable further code-level inspection. Moreover, traditional resource profiling [39, 67] alone, as shown by our evaluation (§3.6.4), is sometimes too coarse-grained to accurately pinpoint the true resource bottleneck.

PerfProbe's cross-layer diagnosis approach aims to address these limitations. It localizes to function calls with running time correlated to the performance variance (a.k.a., *critical functions*), and pinpoints what system resources (e.g., CPU, network, disk, etc.) they interact with cause their running time variance (a.k.a., *relevant resource factors*). In above example, PerfProbe further pinpoints CPU as the resource bottleneck for SCrypt.scryptN and disk I/O for Posix.readBytes (Table 3.2).

3.2.2 Profiling Challenge

Android's built-in profiler *Traceview* [72] (integrated in CPU profiler [51] in latest Android) provides runtime visibility of an app's call stack and can be used for characterizing critical functions. While its sampling mode [51], which captures the call stack at fixed *sampling intervals*, is suggested for reducing the performance impact to apps, if it is kept always-on for profiling an actively used app, the app is likely to become unresponsive and throw an ANR error. Thus, we propose to support event-triggered profiling on only developer-configured user interactions (§3.3.1). Also, our empirical study indicates that small sampling intervals may still introduce high overhead to the runtime execution, especially when it involves CPU or disk I/O intensive workload. Figure 3.2 shows the profiling overhead (in relative increase of latency due to profiling) under different sampling intervals when the computation-intensive optical character recognition (OCR) is performed to extract texts from images in 3 popular apps. First, the 10-95% overhead incurred by small sampling intervals are unacceptable for real-world deployment. Second, this large overhead may skew the running time of function calls and affect the accuracy in pinpointing

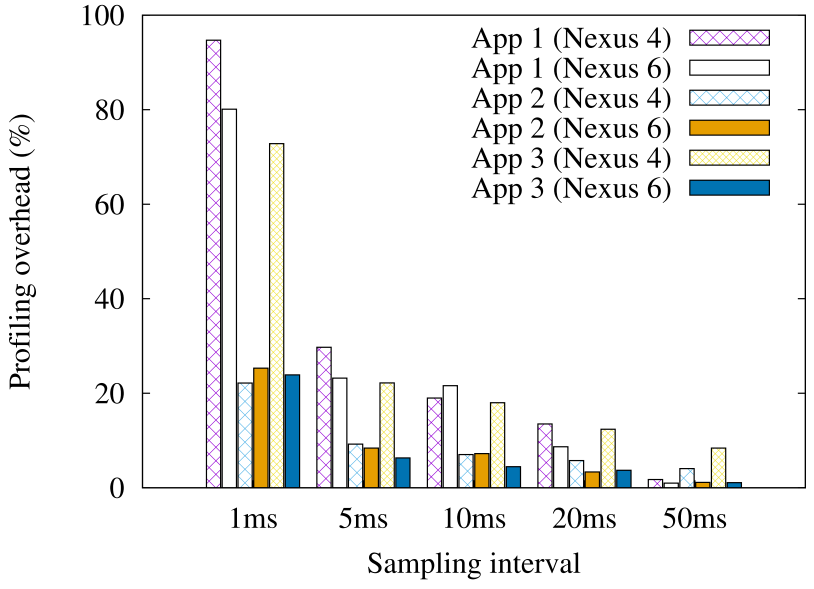


Figure 3.2: Profiling in different sampling intervals and hardware platforms

app-layer execution slowdown [34]. For example, profiling of App 3 on Nexus 4 device incurs 2-3x increase in running time of file operations, causing corresponding function calls (with small running time in reality) to be wrongly identified as execution hotspots.

As Figure 3.2 implies, large sampling intervals may lead to smaller overhead, but by design prevent capturing of function calls completed within a sampling interval and thus hinder fine-grained performance inspection. Moreover, though the profiling overhead commonly decreases when the sampling interval scales up, the performance impact of profiling varies across apps with similar workload and across platforms for a same app. One approach to find a proper sampling interval that preserves sufficient profiling granularity with small runtime overhead is profiling in advance, but becomes hard to scale given the large number of apps and high variety of platforms. To address this challenge, we propose to track the performance impact caused by profiling at runtime and based on which adjust the sampling interval to constrain the profiling overhead to the current app execution below some configurable bound. Our approach is agnostic to apps or platforms and requires no extra manual efforts (§3.3.2).

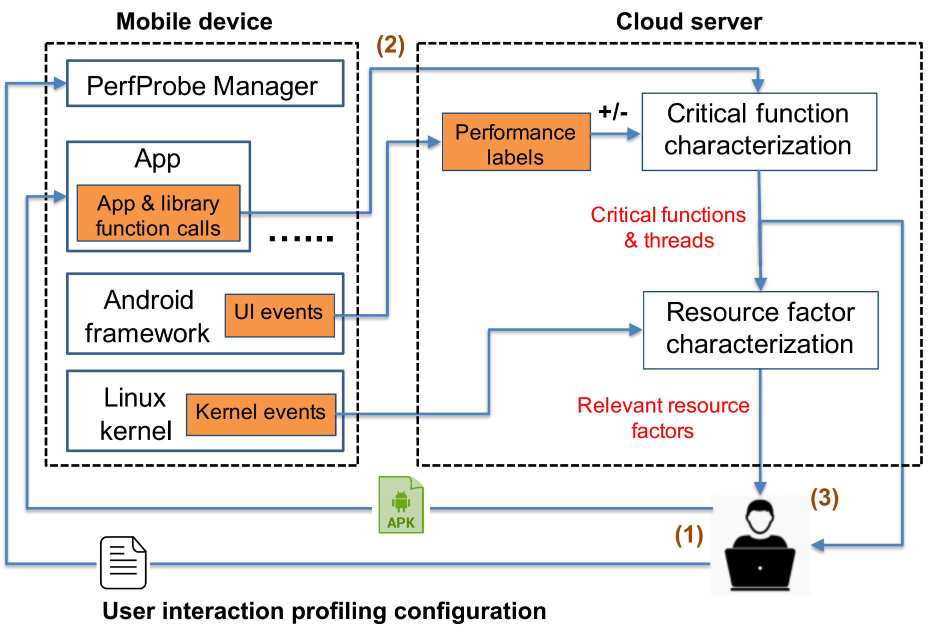

Mobile device
Cloud server
(2)
PerfProbe Manager
App
App & library function calls
......
Android framework
UI events
Linux kernel
Kernel events
Performance labels
+/-
Critical function characterization
Critical functions & threads
Resource factor characterization
Relevant resource factors
APK
(3)
(1)
User interaction profiling configuration

Figure 3.3: PerfProbe overview (“+/-” represents good/bad performance labels)

3.2.3 System Overview

As illustrated in Figure 3.3, PerfProbe consists of two key modules: an **on-device performance monitoring module** (§3.3) and a **problem diagnosis module** deployed on a server (§3.4). Its workflow has following steps: 1) App binaries are installed and targeted user interactions are configured on a rooted mobile device running PerfProbe; 2) The on-device PerfProbe manager in the performance monitoring module controls the profiling of preconfigured interactions and records multi-layer runtime traces, which are periodically uploaded to a remote server (e.g., once per day); 3) The problem diagnosis module analyzes traces to detect unpredictable performance slowdown in a user interaction and if any slowdown is detected performs further diagnosis to provide app developers or device vendors with cross-layer diagnosis insights.

3.3 Performance Monitoring

PerfProbe's performance monitoring module measures the latency of user interactions by instrumenting Android's UI framework to intercept common UI input and update events, and records the app-layer runtime execution using Traceview [72] and system-wide OS events using Panappticon [32], which together form the input to the diagnosis module. To mitigate the runtime overhead caused by this cross-layer monitoring, we make two improvements on existing profiling mechanism in Android.

3.3.1 Event-Triggered Profiling

We instrument UI event handlers in Android's framework to monitor user's invocation on UI components of an app in the run time and start the profiler when certain UI component (e.g., a touch button on a particular view) is invoked. The PerfProbe manager provides an interface for developers to configure user interactions to be profiled, by providing the resource ID (which is device independent and determined at compile time) of the UI components for denoting input and output of an interaction, app package name of an interaction, and profiling parameters including the profiler's sampling frequency and profiling duration. In the run time, when a pre-configured input UI component is invoked, an intent is broadcasted and intercepted by PerfProbe manager, which then launches the profiler based on the configuration. Auxiliary information (e.g., timestamp, location, network trace, CPU load, system log) when profiling an interaction can also be optionally recorded by PerfProbe manager. This asynchronous messaging, by separating the app execution from the profiling process, aims to prevent any stall on the app due to the launch of profiling. One concern with this design is that the profiler may miss some early phase of the app execution, since the app does not wait once sending the intent. Through our empirical study on a wide range of apps (§3.6.2), we validate that profiled events consistently cover key execution of an interaction, since intent messages are received promptly by PerfProbe manager and profiling starts immediately after a user input is performed.

3.3.2 Adaptation of Sampling Intervals

Android's profiler in sampling mode runs as a background thread spawned from an app and periodically (at sampling interval) records the call stack of each thread in an app process sequentially, during which the whole app process (i.e., all its threads) is paused. This pause is the major source of overhead in profiling an app. Due to this design, the pause time depends on the number of threads in an app process and also the running time of the profiler thread, which can be affected by runtime resources of the platform. Our empirical study on apps of different categories shows that the pause time for one sampling may vary from several to hundreds of milliseconds.

Following the intuition that profiling should be made less frequent to cause shorter pause to an app when the app is performing resource-intensive operations, we propose to adjust the sampling interval at runtime based on the pause duration observed in most recent profiling and the computation intensity of current execution in an app. Based on our observation on the source overhead, we define the *relative profiling overhead* (i.e., the percentage of increase in app latency due to pause for profiling) as $O(n) = \frac{P(n)}{S(n)+P(n)}$, where $P(n)$ denotes the observed app pause duration, $S(n)$ denotes the sampling interval for n^{th} profiling round. To limit the profiling overhead, $O(n+1)$, below some configurable bound during the intervals when the profiled app will be experiencing high load, we determine a new sampling interval $S(n+1)$ using the following equations with a user configurable bound, denoted as T ($0 < T \leq 1$). Parameter T in our experiments is set to 0.03. Note that we also need to ensure that the new sampling interval is not shorter than the current pause duration.

$$S(n+1) = \begin{cases} max(S(n), P(n), \frac{P(n)}{T} - P(n)), \text{if high load} \\ max(P(n), min(S(n), \frac{P(n)}{T} - P(n))), \text{otherwise} \end{cases}$$

Following this adaptation model, small sampling intervals (e.g., 1ms) are initialized when profiling starts and the sampling interval is updated after each sampling. In our current design, an app is classified as at high load if its total CPU usage time across multiple cores in the most recent sampling round exceeds the sampling interval.

3.4 Problem Diagnosis

As show in Figure 3.3, the input to the diagnosis module includes function call profiles, UI event logs and OS event traces from deployed devices. User-perceived latencies of an interaction can be determined from UI event logs. Performance labels, with the labeling criteria specified by developers, indicates the occurrence of slowdown in one run based on the distribution of all measured latencies of an interaction. In our evaluation, we use a binary indicator for labeling: given runs of an interaction found with long tail latency doubling or multiplying average latency, any run with perceived latency higher than a threshold is associated with a bad performance label and otherwise a good label. If any unpredictable slowdown is detected, a two-step trace-based diagnosis (detailed as follows) is performed to provide human developers with app and OS-layer diagnostic insights and facilitate root cause identification.

3.4.1 Approach Overview

Running on a cloud server, the diagnosis module performs trace analysis by first zooming into an app-level program execution and then inspecting its interaction with OS in two sequential steps, in order to gain holistic insights on the source of problem at app program level and the cause of problem at system level. Specifically, as illustrated in Figure 3.3, the first step takes the performance labels and app and library function call trace for many runs of an interaction as input, and pinpoints a small subset of functions (a.k.a., critical functions) within the function call trace that are most accountable for the performance variance. For each critical function, its executing thread and time intervals are also generated in the output. The second step leverages the output of the first step and OS event traces as input to extract runtime resource usage features relevant to the execution of each critical function, including but not limited to CPU, network, disk resource usage and IPC usage, and associates each critical function to one or several resource usage features based on correlation explain what causes its execution slowdown. Finally, both the critical functions and their relevant resource factors are presented to human developers to guide their further root cause analysis of the performance variance. We select functions as one diagnosis output because they are program semantic-rich and easy-to-reason for developers, and resource factors as the other because they are usually related to the root cause of performance variance. The remaining subsections present the technical details of this two-step analysis.

3.4.2 Critical Function Characterization

In critical function characterization, a candidate set of critical functions is first selected. Decision tree based learning, taking each run as one data sample, in which the total execution time of each critical function candidate acts as an input feature and the performance label as an input label, is performed to identify a small set of critical functions with execution time correlated to the performance variance. In the generated tree, each node corresponds to a critical function.

Critical function candidate. A *critical function* satisfies the following requirements:

- A critical function consumes a significant amount of execution time of an interaction, based on the intuition that time-consuming functions tend to cause a stronger impact on the user-perceived latency of an interaction.

- The execution time of a critical function varies significantly between runs with different performance labels, based on the intuition that the extra time spent in that function will contribute to the overall user-perceived latency if it causes the performance slowdown.

To fulfill the first requirement, we compute the total time spent in each function for each run based on the function call trace. Then we pick the top-K functions with longest total execution time in each run and merge them across runs to form a candidate set of critical functions. To satisfy the second requirement, we aim to select a small set of functions from the candidate set such that their total execution time in runs with performance slowdown is consistently longer than that in runs without performance slowdown. In other words, we can apply conjunction on this set of functions to discriminate runs with good performance labels from those with bad labels.

Identifying critical functions. We construct a decision tree to understand what functions are most correlated to the performance variance. We use decision trees for two main reasons. First, a decision tree using a compact combination of features selected from a large feature set naturally determines a linear boundary to separate data samples with different labels. Second, a decision tree well depicts the preconditions for performance slowdown: with each node identifying a critical function, given a path from the root node to some leaf containing performance slowdown instances alone, the conjunction of nodes along this path defines a precondition for the slowdown, and the disjunction of all such paths define a set of preconditions under which performance slowdown occurs.

Decision tree details. We use mutual information gain as the criteria for node selection. Before the decision tree characterization, we first apply function pruning by evaluating the relative difference of the total time feature for each function f in the candidate set: prune f if (m_-^f - m_+^f) $<\alpha$ (p_{95}^f - p_5^f). The relative difference is the absolute difference of means over the central range of a feature's values in groups by performance labels (i.e., m_-^f, m_+^f), where central range is the difference between the two 95-percentile values p_{95}^f and p_5^f. α is set as 0.1 by empirical study. The remaining functions in the candidate set provide the input features, which along with performance labels will be used for feature selection to generate splitting nodes of a decision tree. In feature selection, given a set of features with equally highest mutual information gain, we select one with largest relative difference as the splitting node. Moreover, to reduce the variance and avoid overfitting, we

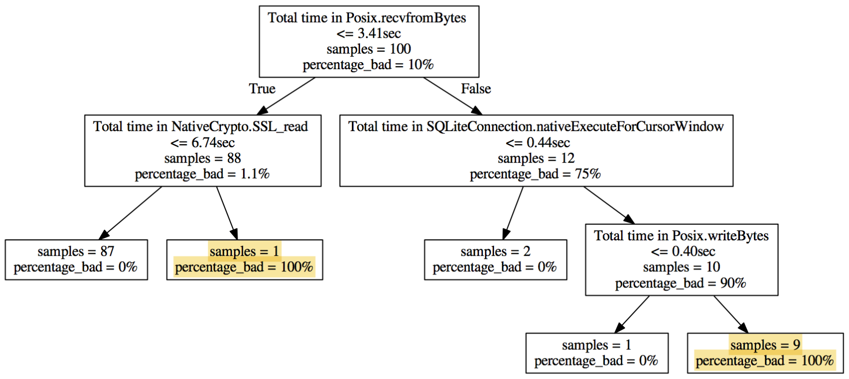


Figure 3.4: Decision tree to characterize critical functions in Vine interaction (two slow-down preconditions as conjunctions of nodes from the root to a highlighted leaf, recvfromBytes and SSL_read identified as critical functions, the other two nodes are pruned because the latency of their corresponding functions is insignificant)

stop generating a splitting node when it reaches certain depth or contains too few samples. Through our empirical study, we find that the depth of a generated decision tree does not usually go beyond 4 when data samples are completely separated. We also exclude a node from being considered as a critical function when it contains too few samples, or when its split gap (i.e., minimum distance between good and bad samples) is not significant, or when the latency of its corresponding function is not significant. We leverage the *scikit-learn* library to implement our decision tree and configure the decision tree to compute *gini* and apply a best split heuristic for feature selection [82].

An illustrative example. We use the *Vine* interaction to showcase our diagnosis flow. From the deployment study, we observe 10 out of 100 runs of *Vine* interaction (listed in Table 3.6) show user waiting 2x long as the median. The decision tree generated from critical function characterization is shown in Figure 3.4 that identifies 2 critical functions, recvfromBytes and SSL_read.

3.4.3 Resource Factor Characterization

The resource factor characterization takes the output of the critical function characterization as input and aims to understand what resource usage causes execution slowdown in each critical function. To achieve that, we first profile the resource usage for each critical function by identifying its execution intervals, under which key resource usage features are extracted. Decision tree learning is then applied on each critical function across runs as data samples, with resource usage features as input features and a binary indicator for the execution time of the critical function as input labels, to identify the resource factors relevant to the slowdown of that critical function.

Identifying relevant execution intervals. Given a set of critical functions, we define a thread executing it as a *critical thread* and its duration as a *relevant execution interval*. Thus, a relevant execution interval corresponds to a critical function and a critical thread. The next step of diagnosis is to narrow down to these execution intervals and reason why slowdown happens in each critical function.

To determine the most relevant resource factors, we construct a set of resource features for each critical function. We summarize key resource features in Table 3.1. Note that the resource usage feature set is extensible for the characterization. For each run, we then sum up each type of resource usage under all relevant execution intervals to form one resource feature for a critical function.

Resource feature	**Description**
CPU usage time	Time spent in running state by thread *T*
CPU wait time	Time spent in the ready state waiting for CPU by thread *T*
CPU frequency	Average CPU frequency when thread *T* is running
Interruptible sleep time	Time when thread *T* is in interruptible sleep
Uninterruptible sleep time	Time when thread *T* is in uninterruptible sleep
Network blocking time	Time when thread *T* is blocking for network I/O
Disk blocking time	Time when thread *T* is blocking for local disk I/O
IPC wait time	Time spent in inter-process communication by thread *T*

Table 3.1: Resource usage features

Extracting resource usage features. To compute how much time is spent in CPU running state, in interruptible/uninterruptible sleep state or in the ready state waiting for context switch-in, we rely on the context switch events within a relevant execution interval. To extract network or disk blocking time, we analyze the I/O blocking events within a relevant execution interval. To obtain IPC wait time, we compute the waiting time between each binder request and response within a relevant execution interval. We also compute the average CPU frequency when a thread is occupying CPU across its relevant execution intervals based on frequency governor events.

Pinpointing relevant resource factors. The resource factor characterization of each critical function is also achieved through decision tree learning similar to that in the critical function characterization, in which a corresponding node for a critical function contains a subset of runs that becomes input samples to the characterization at this step. The input features of each sample consists of the key resource usage features for a critical function. In the critical function characterization, a threshold on execution time has also been determined for each critical function, which is used for labeling the input data in this step. In other words, the label indicates whether execution slowdown occurs in a critical function. The same feature pruning and node selection technique applied to critical function characterization are followed to construct a decision tree, in which each node identifying a resource usage feature relevant to the slowdown of a critical function. Using the example in Figure 3.4, all 100 samples (88 positive, 12 negative) are used to characterize the resource factor for recvfromBytes, resulting in a decision tree with the network blocking time as its root (i.e., relevant resource factor). Analysis of 88 samples in the left branch (87 positive, 1 negative) pinpoints interruptible sleep time as the relevant resource factor for SSL_read.

3.5 Implementation

We have implemented a PerfProbe prototype that can be deployed on Android 4.4.4 KitKat and 5.1.1 Lollipop. While we build on top of state-of-the-art tracing and profiling tools to implement system-wide, cross-layer instrumentation (as illustrated in Figure 3.3), we find that additional automation features are necessary to support our usage scenario. Also, low-overhead profiling and tracing techniques are essential to support performance diagnosis in the wild. We highlight several implementation challenges from both perspectives as follows.

System instrumentation and trace collection. In our prototype, OS-layer event tracing is built based on Panappticon [32] that was originally developed for Android 4.1. Furthermore, we reuse Android's built-in method profiler (a.k.a., Traceview) to implement our app profiler. To integrate them into our prototype for performance diagnosis, we make the following implementation efforts. First, to deploy Panappticon on newer Android systems, we port its kernel and framework-level instrumentation into the kernel and framework code base for KitKat and Lollipop (i.e., AOSP 4.4.4 and 5.1.1). Second, while the Android profiler is accessible from the Traceview GUI, to support automatic profiling using the Android profiler, one feasible way is to leverage the *am* command line interface to control the profiler. However, we find that the *am* command line interface in existing AOSP can only launch the profiler in tracing mode, which incurs large performance impact to the app and is not an option in PerfProbe, we extend the profiler implementation across the Dalvik VM and framework layer of AOSP to support programmable sampling-based profiling. Third, our prototype includes a system app (a.k.a., PerfProbe manager) that runs a long-living background service for not only collecting OS event and function call traces from memory buffer and uploading them to a preconfigured server if the device is under certain network and/or battery condition, but also enabling programmable control of Android's profiler to minimize its impact on app performance (detailed in the rest of this section).

Event-triggered profiling. Another key challenge is to enable low-overhead tracing and profiling based on pre-configured user interactions in the wild. One naive approach is to keep the tracing and profiling process long-running once the system is up. Our empirical study shows that, though the OS event tracing can keep running in the background with little resource overhead, if the app profiling lasts for a long period of usage, most apps will become unresponsive (i.e., throwing an ANR error). Therefore, our prototype needs to support event-based profiling, meaning that the profiling is activated only when the interested interactions are performed. To achieve that, we instrument UI event handlers in Android's framework to monitor user's invocation on UI components of an app in runtime and start the profiler when certain UI component (e.g., a touch button on a particular view) is invoked. The PerfProbe manager provides an interface to configure user interactions to be profiled. The configuration includes the resource ID of its input UI component and package name as identifiers of an interaction, as well as the sampling frequency and profiling duration. In runtime, when a pre-configured UI component is invoked, an intent is broadcasted and intercepted by PerfProbe manager, which then launches the profiler based on the configuration. This asynchronous messaging, by separating the app execution from the profiling process, aims to prevent any stall on the app due to the launch of profiling. One concern with this design is that the profiler may miss some early phase of the execution. Through our empirical study on a wide range of apps, we validate that profiled events consistently cover key execution of an interaction, since intent message are received promptly by PerfProbe manager and profiling starts immediately after a user input is performed.

3.6 Evaluation

We perform controlled experiments on 5 real Android apps with synthetic performance variance introduced by perturbing different system resources or triggering a programming mistake into app source code (summarized in Table 3.2) in some runs. Diagnosis results demonstrate that our diagnosis approach can always localize functions with injected faults or correctly pinpoints the perturbed system resource.

We also conduct a real-world deployment and diagnosis study to answer following questions. How useful is PerfProbe in guiding root cause diagnosis and code fixing of unpredictable performance issues for real-world app developers (§3.6.1)? How effective is PerfProbe in diagnosing real-world unpredictable performance issues with popular An-

App	Interaction	Injected problem	Critical function	Relevant resource factor
Download Manager [31]	Download a file	Network delay	`Posix.recvfromBytes`	Network blocking time
H&M [46]	View an item			
CNET [27]	View a post			
Sudoku Solver [56]	Solve a grid	Programming flaw	`SudokuCore.solveMethodOptimised`	CPU usage time
SSE [86]	Encrypt a file from SD card	Background load	`SCrypt.scryptN`	CPU wait time
		Throttled disk access	`Posix.readBytes`	Disk blocking time

Table 3.2: Diagnosis results for synthetic performance issues

droid apps (§3.6.2 & §3.6.3)? What benefit can PerfProbe's cross-layer characterization achieve compared to existing diagnosis approaches (e.g., monitoring system calls, resource profiling) (§3.6.4)? How much overhead can PerfProbe incur to a mobile device and how much performance impact can our adaptive sampling reduce (§3.6.5)?

3.6.1 Android App Developer Study

We apply PerfProbe to diagnose user-reported performance problems in 6 open-source Android apps (user rating above 3.5 and over 50K downloads) and report our findings to their developers for feedbacks. Specifically, we mimic app developers to conduct followup debugging based on the critical functions and relevant resource factors output from PerfProbe. To quantify the extra manual effort for code inspection with hints from PerfProbe, we define a metric *relative extra effort* based on prior works [156, 166], as the ratio of the portion of app source code we manually inspected based on the critical functions from PerfProbe to the portion of app source code invoked in the run time (i.e., baseline).

Table 3.3 shows the relative extra effort and summarizes the root cause findings (detailed in app's GitHub issue tracker) for each issue. With PerfProbe, less than 3% of the executed source code needs to be inspected and the pinpointed resource factors give direct explanation to the running time variance in pinpointed critical functions. We reported our findings through GitHub's issue tracker to app developers and obtained acknowledgment from developers of 3 apps (highlighted in Table 3.3). Based on followup analysis using PerfProbe's output, we also suggested feasible optimizing solutions to some issues. The **iNaturalist** and **Riot** developer invited us to submit pull requests for our suggested solutions. Our suggested strategy for iNaturalist [83], implemented in 90 lines of code, has been adopted by its developer (detailed as follows).

App	Interaction	Root cause summary	Relative extra effort
iNaturalist [47]	Click Guides tab	Overloaded sequential web requests [83] $*$	0.45%
Riot [74]	Open chat room	Web requests and computation delay for bitmap decoding [87] $*$	2.43%
K9 Mail [57]	Sync mailbox	Occasional loss and re-establishment of IMAP connection [55] $*$	0.70%
c:geo [24]	Search nearby cache	Delay for sequential web requests [53]	0.33%
GeoHash Droid [38]	Launch app	Location query and computation delay for map rendering [88]	2.78%
Tomahawk Player [99]	Search songs keyword	Web server unavailability [54]	0.81%

Table 3.3: Summary of diagnosis reporting: $*$ indicates our report is acknowledged by app developer

Case study. iNaturalist app (over 500K downloads in Google Play) enables users to view or upload plant and animal observations. PerfProbe pinpoints Posix.recvfromBytes (invoked by getAllGuides) as the critical function, with network blocking time as its relevant resource factor. To trace the source of network blocking, we investigate the definition of getAllGuides and discover that a series of HTTP requests are issued sequentially to retrieve many ($>$1000) JSON objects. As we observe that users cannot view all loaded items from the UI screen, we suggest limiting the number of JSON objects to be retrieved through HTTP requests and adding a "Load more" option in the UI for users to choose whether to continue loading more new items in order to reduce the user waiting time for UI update [83]. We implemented our suggested strategy [83] by adding 90 lines of code and validated that it reduces the user-waiting time for this interaction by 86%. Eventually, the developer adopted our suggestion and changed the app to load only the first page of items for better interactive experience [33].

Riot. Riot is a popular group chatroom app with an average rating of 4.5 in Google Play. Besides delay for web requests, PerfProbe also pinpoints `nativeDecodeAsset` (Android's bitmap API) that is invoked by `setContentView` as a critical function, caused by longer CPU wait time (i.e., computation bottleneck). We investigate the resource file with `setContentView` in app source code and find the resource file includes several large bitmaps to be decoded, which is usually memory-heavy and computation-intensive, for rendering the layout of the chatroom activity. To alleviate this computation bottleneck, we suggest loading a scaled-down version of bitmap following Android's developer guideline [60] for accelerating the decoding. Riot's developer acknowledged our findings and waits for our pull request for our suggested fix [87]. As unveiled from our diagnosis, the performance of `nativeDecodeAsset` is dependent on the size of input bitmap and thus solely pinpointing the critical function alone is not adequate for root cause understanding. PerfProbe correspond with its resource bottleneck and successfully leads to more insights about root cause.

3.6.2 Real-World Deployment

To diagnose unpredictable performance slowdown issues in the real world, we deploy PerfProbe's monitoring module on Nexus 4 and 6 devices to monitor common user interactions for a wide range of popular Android apps. We select top-ranked Android apps [16, 40] from different categories and obtain in total 100 popular apps (summarized in Table 3.4). For each app, we identified a common interaction based on our domain knowledge of an app and configure PerfProbe manager to monitor it. In each deployment run, a subset of selected apps were installed on a test device and replaced by another subset in the next run. To mimic real-world daily usage, a device was brought to different locations, including an office, campus, and residential environment (all with WiFi access), where UI inputs were automatically replayed using `UIAutomator` [100] to launch an interaction randomly picked from the preconfigured ones. During the deployment, each preconfigured interaction was performed for sufficiently many times (65$\sim$110 runs).

Out of the 100 apps, we discover 11 apps (spanning 6 main categories in Table 3.4) in which tail latencies are 1.5$\sim$8x as long as the median latency. Figure 3.5 shows the distribution of waiting time for the user interactions of these apps (listed in Table 3.5). Root cause analysis based on PerfProbe's cross-layer trace diagnosis is presented in §3.6.3.

Category	# of apps	Downloads
Tool	32	500K∼500M
Shopping	15	10M∼500M
News	10	50K∼1B
Social	8	50M∼5B
Media	6	100M∼500M
Navigation	5	100M∼5B
Other	24	50K∼500M

Table 3.4: Android apps for performance monitoring in the real world

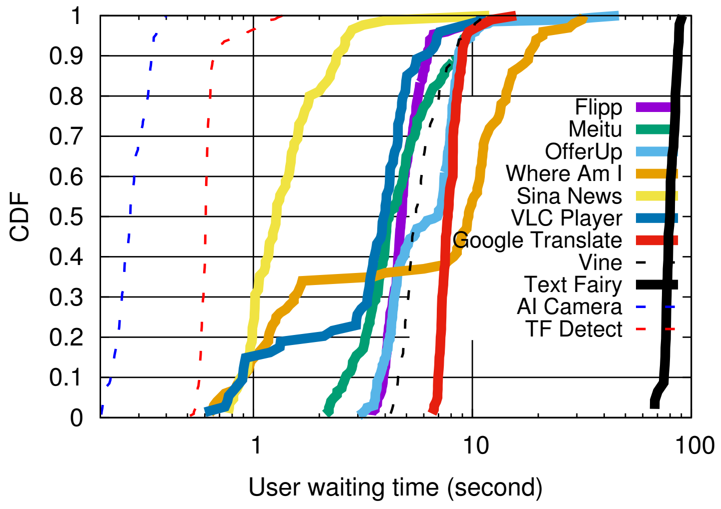


Figure 3.5: Unpredictable performance slowdown in popular Android apps

3.6.3 Diagnosis of Performance Issues

We apply PerfProbe to diagnose performance variance (listed in Table 3.5) uncovered in our deployment study (§3.6.2). We label a run as performance slowdown if its waiting time is longer than the median waiting time by one or two standard deviation, depending on the skewness of the distribution of the user waiting time. Table 3.6 summarizes the diagnosis output of PerfProbe for each case, with their diagnosis details documented in an anonymous website [70]. We also perform further analysis and validation based on PerfProbe's diagnosis findings as follows.

- For CPU frequency bound, by reconfiguring existing userspace parameters of the Dynamic Voltage and Frequency Scaling (DVFS) governors [59], the tail latency of

App	Interaction	Version
Vine [102]	Launch app to play a video	5.18.0
Flipp [35]	Launch app to load flyers	5.0.1
OfferUp [66]	Launch app to load deals	2.2.25
Sina News [81]	Click a bookmarked post	4.9.5
Google Translate [42]	Translate texts in an image	5.5.0
VLC Player [103]	Play a video from SD card	2.0.6
Meitu [61]	Enhance a photo in SD card	5.1.9.1
Where Am I [107]	View current address	1.14
Text Fairy [93]	Extract texts in an image	3.0.8
AI Camera [1]	Detect objects in camera	Demo
TF Detect [94]	Detect objects in camera	1.0.0

Table 3.5: Android apps with unpredictable performance slowdown

3 CPU-bounded interactions are reduced by 32-40%.

- For disk I/O factor, by applying a common tweak of a system parameter to boost the access speed of on-device SD card, the tail latency of 2 disk-bounded interactions is reduced by near 50%.

- For network or server-side factor, we investigate network trace captured by *tcpdump* for further validation.

- For GPS problem, we trace the destination of the pinpointed inter-process communication to locate GPS-related system process.

3.6.3.1 DVFS governor issue

This case study presents diagnosis and validation on 3 apps where performance is affected by the computation speed controlled by DVFS governors.

Problem diagnosis. When performing offline OCR-based text extraction using *Text Fairy* on a Nexus 4 device (1.5 GHz quad-core Krait), as shown in Figure 3.5, a user may wait extra 20 seconds (compared to 70 seconds in fast runs) for English texts to be extracted from an image. PerfProbe identifies TessBaseAPI.nativeGetHOCRText, defined in

App	Critical functions	Relevant resource factors	Root cause summary

Vine	`Posix.recvfromBytes`
	 `NativeCrypto.SSL_read`
		Network blocking
		 Interruptible sleep
			network or server-side delay

Sina News	`Posix.recvfromBytes`	Network blocking	network or server-side delay
Flipp OfferUp Google Translate	`NativeCrypto.SSL_read`	Interruptible sleep	network or server-side delay

VLC Player	`VideoPlayerActivity.onCreate`
	 `MediaCodec.start`
		Disk blocking
		 Disk blocking
			Slow SD card read speed

Meitu	`SmartBeautifyActivity.onCreate`	Disk blocking	Slow SD card read speed
Where Am I	`MessageQueue.next`	IPC wait	GPS signal locking delay
Text Fairy	`TessBaseAPI.nativeGetHOCRText`	CPU frequency	Frequency capped by governor policy
AI Camera	`classificationFromCaffe2`	CPU frequency	Frequency capped by governor policy
TF Detect	`org.tensorflow.Senssion.run`	CPU frequency	Frequency capped by governor policy

Table 3.6: Diagnosis output for real-world performance slowdown

Google's Tesseract OCR API [92] and invoked by a worker thread, as a critical function for all slowdown instances. Furthermore, the average CPU frequency is pinpointed as its relevant resource factor. Based on the split threshold from the resource factor characterization, the average frequency along the execution of the critical fuction reaches above 1.2GHz for all fast runs. Figure 3.6 shows the time series of the CPU frequency for the core on which the critical function is executed for a randomly picked fast and slow run in traces collected from our real-world deployment. We can clearly see that the frequency scaling gets stuck at 1.1GHz in some runs when executing the critical function and thus leads to slowdown of the overall interaction.

Root cause validation. To validate the root cause in frequency governor, we intentionally increase the upper frequency limit for scaling (i.e., *scaling_max_freq*) to 1512MHz (maximum available scaling frequency on Nexus 4) and also change the governor type from

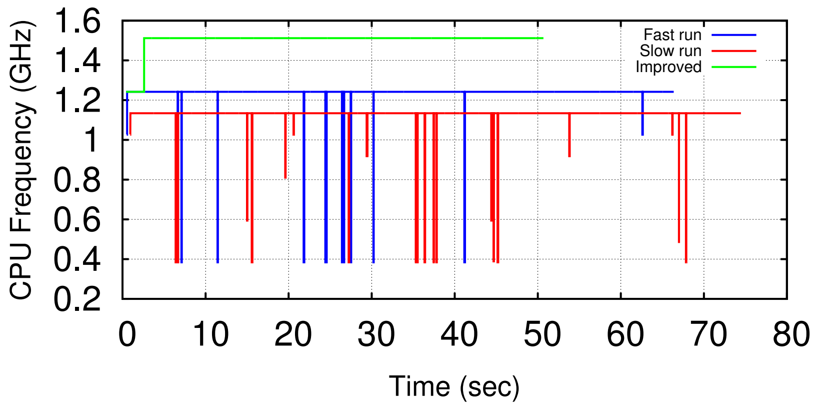


Figure 3.6: CPU frquency for executing critical function over time

ondemand (default governor type in Nexus 4) to *performance* right before an interaction is performed. As a result, the tail user waiting time is reduced by 40% (to 53 seconds). Interestingly, even when the *ondemand* or *interactive* governor is used, the tail latency can be reduced to 57 seconds by presetting *scaling_max_freq* to 1512MHz. Figure 3.6 indicates the execution time for the critical function is significantly reduced when its execution finishes at maximum frequency. Note that this strategy is unrealistic as a long-term strategy given the energy and temperature constraint.

Object detection apps. Our diagnosis on *AI Camera* and *TF Detect* also reveals CPU frequency as the resource bottleneck for running the pinpointed critical functions for detecting objects in a camera frame on a Nexus 6 device (2.7 GHz quad-core Krait 450). Further investigation on the CPU states leads us to the interactive frequency governor policy: *scaling_max_freq* is capped at 1958MHz when either app is running. To improve their object detection performance, we initialize *scaling_max_freq* as 2649MHz (maximum available scaling frequency on Nexus 6) and set *ondemand* governor at app launch. As a result, the per-frame object detection latency is reduced by 32% on **AI Camera** and 40.6% on **TF Detect**.

3.6.3.2 Disk hardware issue

This case study presents diagnosis and validation on 2 apps with performance degradation caused by disk I/O.

Problem diagnosis. As shown in Figure 3.5, the latency for playing an HD video (of size 35.5MB) from SD card in *VLC Player* or processing a photo (of size 1.93MB) in SD card in *Meitu* can take more than 2.5x of median (both around 4 seconds) on a Nexus 4 device. PerfProbe reveals that 62.5% of slowdown instances are characterized by the critical functionVideoPlayerActivity.onCreate on the main thread to load the video playing activity, while the rest happen in executing the other critical function MediaCodec.start on the VLC object thread. Both are bounded by disk blocking. Similarly, PerfProbe pinpoints the critical function SmartBeautifyActivity.OnCreate and attributes its slowdown to slow disk I/O for the photo enhancing interaction in *Meitu*.

Root cause validation. To validate the disk I/O bottleneck, we increase the SD card access speed for Android devices by tuning the read-ahead buffer [48, 50, 49]. The read-ahead buffer defines the size of a disk block to be loaded into memory for each read. For long sequential file read operation (e.g., copying large file from SD card), having a larger read-ahead buffer will usually speed up the read process. We find its size is set to 128KB by default on Nexus 4 phones and can be configured through the *sysfs* interface. Through empirical tuning, we find the SD card read speed on a Nexus 4 phone is improved significantly as the size of read-ahead buffer increases from 128KB to 2048KB. Therefore, we reconfigure the read-ahead cache size as 2048KB on our Nexus 4 test device, while keeping the other setup unchanged, to perform controlled testing on both interactions. Figure 3.7 shows the improvement of user waiting time for both apps after increasing the read-ahead cache. Specifically, the tail user waiting time is reduced by 45% (to below 6 seconds) for **VLC Player** and by 42% (to below 7 seconds) for **Meitu**.

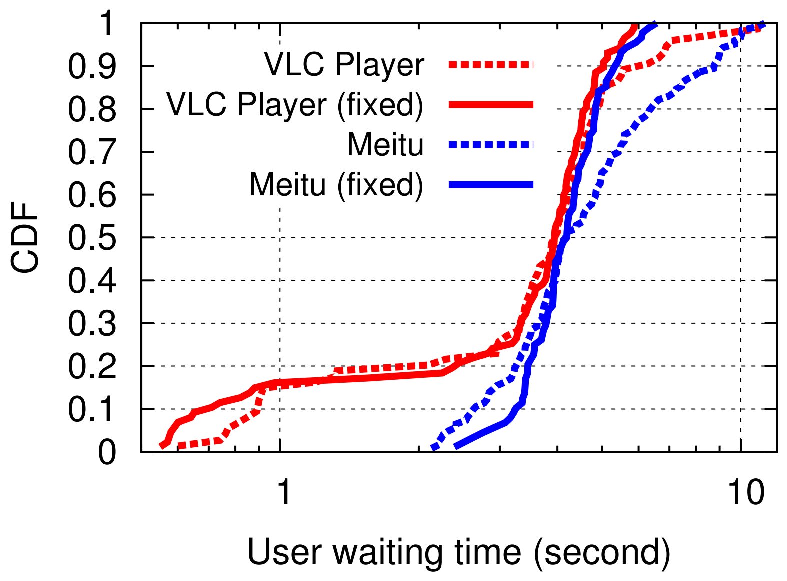


Figure 3.7: Performance improvement by mitigating disk I/O bottleneck

3.6.4 PerfProbe's Benefit Highlight

One baseline for performance diagnosis is identifying resource bottleneck from the overall resource usage throughout the whole execution of an interaction. This approach leads to misidentification of resource bottlenecks in 8 out of 22 cases, including **Text Fairy**, **AI Camera**, **TF Detect** (true cause is the CPU frequency cap set by the DVFS governor), **VLC Player** (true cause is disk I/O delay), **Where Am I** (true cause is GPS handling delay), **Riot** (true cause is server-side delay and waiting time on CPU resource) and **K9 mail**, **iNaturalist** (true cause for both is server-side delay). Therefore, critical function characterization is indeed necessary for achieving high accuracy in pinpointing relevant resource factors.

Another baseline approach is monitoring system calls. For issues due to the DVFS governor, while system calls can hardly reveal frequency change on different CPU cores, identifying the computation bottleneck caused by frequency governor can be inaccurate even when general resource profiling is used. For issues due to the disk I/O bottleneck, we try profiling system calls using *strace* on both cases to check if the run time of disk-related system calls has significant variance to account for the bottleneck, but find that the total run time spent in system calls for both cases take up less than 5% of the overall latency and that disk-related calls do not show significant variance in the run time.

3.6.5 Runtime Impact & System Overhead

App performance impact. To evaluate the benefit of our adaptive sampling mechanism for app profiling, we conduct a controlled experiment to measure profiling impact on app performance (i.e., increase of user-perceived latency in PerfProbe) under adaptive and fixed 10ms/20ms sampling interval (baseline). Compared to fixed sampling intervals, our adaptive sampling mechanism increases the sampling interval of Traceview when resource-intensive operations are ongoing for some apps and converges at a larger interval to maintain low runtime overhead. The sampling interval decreases and converges to 5-50ms once those expensive operations finish. For interactions in Table 3.2, 3.3, 3.6, adaptive sampling incurs at most 3.5% increase of the median latency of an interaction, while fixed sampling intervals incur 3-22% increase. Note that this 3.5% increase causes negligible effect to detecting and diagnosing performance slowdown with at least 50% increase of the overall latency. Though adaptive sampling may miss function calls with small running time due to reduced sampling frequency for accommodating resource-intensive operations, the output critical functions and relevant resource factors for all studied interactions remain consistent with fixed sampling intervals and adaptive sampling, mainly because only top-K time-consuming functions are taken as input for critical function characterization.

CPU & memory overhead. In our current prototype, PerfProbe manager uses a 10MB memory buffer for logging OS kernel events, a 15MB buffer for Android framework events and Traceview's default 8MB buffer for an app's call stack. PerfProbe showed no noticeable increase in CPU or memory usage in our deployment. We also measure the logging time (averaged over 100K executions): logging a kernel event takes less than 1 microsecond and logging a framework event takes 3.2 microseconds in an instrumented function.

Storage & energy overhead. In our current prototype, function call and OS event traces are stored in the local storage of a device and periodically uploaded to a remote server when certain network (e.g., in WiFi network) and/or battery condition (e.g, charging state) is met. We conduct a controlled measurement on a Nexus 4 device: based on the PhoneLab [71] data we find that 85% of the time real users perform no more than 210 interactions per day, so we replay 210 interactions on the 11 apps with performance slowdown uncovered from the deployment study (§ 3.6.2) using `UIAutomator` [100] for 5 times with or without PerfProbe enabled, and measure the average storage and energy overhead caused by PerfProbe. Measurement results show that each interaction incurs 10KB$\sim$500KB function call trace and on average 2.2MB OS event trace. Given the growing capacity of mobile device storage and high availability of WiFi networks for trace uploading, this storage overhead is acceptable for real-world deployment of PerfProbe. Moreover, PerfProbe incurs only 1.9% energy overhead to a smartphone device.

3.7 Discussion

We discuss the scope and limitations of PerfProbe and the threats to the validity of our experiments. First, PerfProbe's approach is general to other mobile platforms, e.g., iOS using Instruments [52]. Second, current PerfProbe targets at performance slowdown occurring occasionally. To diagnose slowdown that consistently occurs, additional mechanism, such as resource amplification [168, 205, 195], can be leveraged to amplify resource-intensive operations for enabling PerfProbe's differential analysis. Third, as the output of PerfProbe, the critical functions and relevant resource factors facilitate the root cause analysis of performance issues and may not be the actual root cause. Forth, the synthetic performance issues (Table 3.2) are mainly for validating the accuracy in pinpointing critical functions and relevant resource factors, but the injected performance causes are ell encountered in our real-world app deployment and effectively pinpointed by PerfProbe.

3.8 Summary

PerfProbe is a mobile performance diagnosis framework that associates app and OS-layer runtime information in a lightweight manner to provide holistic, cross-layer insights to the root cause of unpredictable performance slowdown in real-world usage. PerfProbe effectively pinpoints code-level or system resource-layer factors for performance slowdown in 22 Android apps and guides real-world Android developers' code-level fixing to significantly improve app responsiveness.

CHAPTER IV

AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems

4.1 Introduction

The world is facing an enormous revolution of transportation with the emergence of connected and autonomous vehicles. Autonomous driving hold the promise to improve road safety and significantly improve transportation mobility efficiency in our daily lives. As a result, advanced autonomous driving algorithms and software are gaining importance. Autonomous vehicle (AV) systems are being developed and deployed in real vehicles [106, 78, 97, 15] and have demonstrated great promise towards full autonomous driving in the near future. Despite this rapid development, AV systems are facing a number of cybersecurity threats, for example, attacks on automobile Electronic Control Unit (ECU) [162, 125, 144, 44] and key sensing devices for autonomous [174, 36, 63]. However, a highly critical attack surface is still underexplored so far: the AV software systems for making autonomous driving decisions. Since these decisions have direct impact on road safety, it is necessary to understand potential security vulnerabilities in the design and implementations of AV software systems, and proactively address them in the AV system development stage.

We observe that AV software systems are usually composed by a number of key self-driving modules, which interact through *a publish-subscribe communication model* to exchange computation states using different types of messages defined by AV developers. Modules are granted publish or subscribe permission to be a publisher or subscriber for certain types of messages. To improve the functionality and reliability of autonomous driving, these modules are becoming feature rich and as a consequence the messages also become increasingly complex. Based on our empirical study on two popular AV software platforms *Baidu Apollo* [14] and *Autoware* [18], we observed 60-80 types of publish-subscribe messages, each consisting of several to dozens of fields. Despite the new functionality enabled by the rich set of fields, this complex message structure also introduces a new security problem to the publish-subscribe messaging system in an AV system. Specifically, in both *Baidu Apollo* and *Autoware*, we found a number of code examples indicating a common *overprivilege problem* with this messaging model, stemming from a lack of sufficient granularity when granting publish or subscribe permissions of key messages to a module.

Through further in-depth study on both AV systems, we discover two common types of overprivilege in its publish-subscribe messaging, when either: 1) fields in a published message are not used in a particular subscriber; 2) the values of certain fields in a published message are directly copied from other messages subscribed to by the publisher. We characterize these behaviors as subscriber- or publisher-side overprivilege in AV systems, respectively, since the granted publish or subscribe permission of a message to a module that does not follow the least-privilege principle at the message field granularity. We have constructed several concrete exploits of such non-compliance, which demonstrates that this problem indeed exposes a new attack surface to AV systems and may lead to vehicle collision and identity theft for AV owners under a realistic threat model (detailed in §4.3) inspired by existing automotive attack surface analysis. Therefore, we argue that this publish-subscribe overprivilege problem should be fully addressed when designing secure AV software. In particular, to overcome this problem, the publish/subscribe permission should be defined at a message field granularity. Subscribe permission for a field should be granted to a subscriber only when it uses that field for computation, and publish permission should be granted to a publisher only when it modifies the state of that field before publishing.

However, enabling such fine-grained permission control can be challenging in AV software systems, because AV software development is a multidisciplinary task and typically conducted by a large team of developers with different domains of expertise. Moreover, some AV systems are built upon an open platform to encourage open-source contribution of self-driving algorithms and code. As a result, AV system designers may not have complete knowledge about the exact usage of message fields in a module and tend to include as many fields in each message as possible to simplify software development. Even if an explicit message field-level permission model and static access control policy enforcement are enabled in AV software, we cannot fully trust a module to comply with the enforcement at run time, since an AV module can be compromised and the pre-defined access control logic can get bypassed. Because the runtime policy enforcement will be performed on every published or subscribed message, the run time enforcement must incur little overhead. To address these challenges, we propose a systematic analysis and mitigation approach to address this overprivilege problem for AV software.

To effectively detect and mitigate publish-subscribe overprivilege in AV systems, we propose AVGuardian, consisting of a static analysis tool that systematically detects overprivilege instances in AV software and generates the corresponding access control policies at the message field granularity, along with a runtime policy enforcement mechanism to perform online policy detection and prevention. Our static analysis approach handles complex real-world C++ source code, including virtual functions and asynchronous programming models, to both achieve high precision in overprivilege detection and prevent under-granting publish/subscribe permissions. Our runtime policy enforcement can defend against publish-subscribe overprivilege with a single module compromised, and does not require any additional efforts from AV software developers or changes to the AV software development process. As we observe that several popular AV software systems [14, 18] are developed on top of ROS [75], we prototype the policy enforcement component of AV-Guardian as a ROS plug-in that is transparent to AV modules. Trace-based performance evaluation in realistic setup shows that the runtime policy enforcement incurs only 10-millisecond increase of the end-to-end delay for AV's control decision making and does not affect original decision logic.

We performed responsible disclosure and received confirmation from the Apollo developer team that our attack findings are valid under our threat model, and the publish-subscribe overprivileged attack is indeed a general security challenge in AV software development. They also commented that it can be highly beneficial to have a systematic approach to automatically uncover and prevent overprivilege problems, which is exactly the research goal in this work.

The contributions of this work are as follows:

- We discover the overprivilege problem in publish-subscribe messaging model for AV software systems, and perform the first characterization and systematic study. To demonstrate the severity of such problem, we construct three concrete attacks by exploiting vulnerabilities resulting from overprivilege problems in GNSS and LiDAR driver modules.

- We design and implement a data-flow analysis tool to help AV developers perform static detection of publish-subscribe overprivilege problems in AV software and generate fine-grained permission control policies at the message field level to mitigate the security consequence from overprivilege. From runtime profiling results, we observe zero false positives in overprivilege detection and less than 1.8% false negative rate. Using this tool, we are able to automatically detect 523 subscriber-side overprivilege instances and 56 publisher-side overprivilege instances in the Baidu Apollo code base.

- We design an efficient and module-transparent policy enforcement solution to perform online detection and prevention of violation of permission control policies for publish-subscribe communication in ROS-based AV systems. We prototype this solution in ROS and find that it incurs very low overhead, i.e., only 9-millisecond increase in end-to-end delay in Baidu Apollo, and does not affect original AV decision logic.

4.2 Background & Motivation

This section presents a background overview of common AV software systems and how the publish-subscribe messaging model is used to enable interaction among key self-driving modules. We then present our discovery of overprivilege in this messaging model from our study on representative AV software systems (Baidu Apollo [14] and Autoware [18]) and characterize two common types of overprivilege in the publish-subscribe messaging model through a real-world example. Finally, we contrast the publish-subscribe overprivilege addressed in our work with previous approaches to overprivilege in other contexts.

4.2.1 AV Software System

The architecture of most existing AV software systems falls into two categories: model-based [14] and end-to-end [121]. Our study focuses on model-based systems since such designs have already been adopted in many real-world AV systems [15, 18]. Figure 4.1 shows a typical pipeline in model-based AV systems, which often contain a set of modules for performing key self-driving functionalities including vehicle localization, routing, obstacle perception and prediction, path planning, and control decision execution. AV software also often contains driver modules for peripheral sensor devices, such as GNSS, LiDAR, radar, and cameras. Similar to the architecture of ECUs and CAN bus in commodity automobiles, these modules are instantiated as nodes (each in separate processes) that run on a middleware such as ROS [75] and communicate through a publish-subscribe

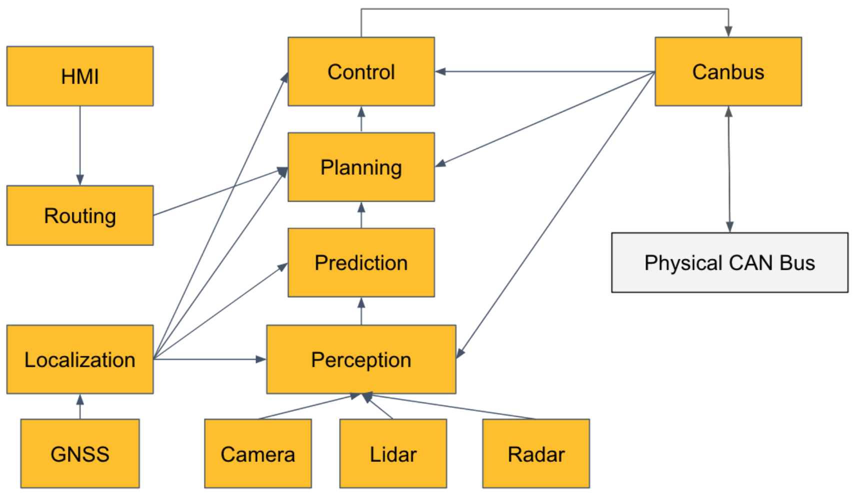


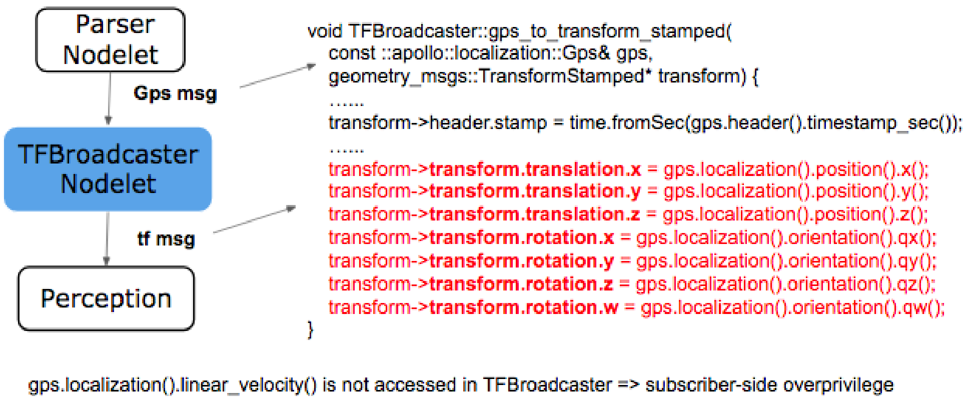
Figure 4.1: Typical architecture of AV software systems (based on Baidu Apollo): rectangles representing a ROS node/nodelet, arrows representing the ROS message flow through ROS's publish-subscribe messaging model.

message channel, acting like a virtual CAN bus for AV systems. In such message channels, the producer of a message is called a *publisher* and the consumer is called a *subscriber*.

In the context of AV systems, the computation pipeline of AV software is dictated by the publish-subscribe message flow. Sensing input from peripheral devices is processed, module-by-module, until a final control decision is reached and executed on the physical actuators. Thus, the messages sent between these modules are responsible for mission-critical communication in AV systems, which directly influence the end-to-end self-driving decisions. Therefore, safeguarding the messaging channel in AV systems is critical to ensure secure and safe autonomous driving.

4.2.2 Publish-Subscribe Overprivilege Problem

Due to the high importance of these messages, the security of this publish-subscribe message channel has already attracted attention of the research community. The state-of-the-art solution, SROS (Security Enhancements for ROS) [84, 199], defines a message-level publish-subscribe permission model and authentication mechanism to enhance the security of publish-subscribe messaging in ROS. However, we find that such message-



```
void TFBroadcaster::gps_to_transform_stamped(
    const ::apollo::localization::Gps& gps,
    geometry_msgs::TransformStamped* transform) {
    ......
    transform->header.stamp = time.fromSec(gps.header().timestamp_sec());
    ......
    transform->transform.translation.x = gps.localization().position().x();
    transform->transform.translation.y = gps.localization().position().y();
    transform->transform.translation.z = gps.localization().position().z();
    transform->transform.rotation.x = gps.localization().orientation().qx();
    transform->transform.rotation.y = gps.localization().orientation().qy();
    transform->transform.rotation.z = gps.localization().orientation().qz();
    transform->transform.rotation.w = gps.localization().orientation().qw();
}
```

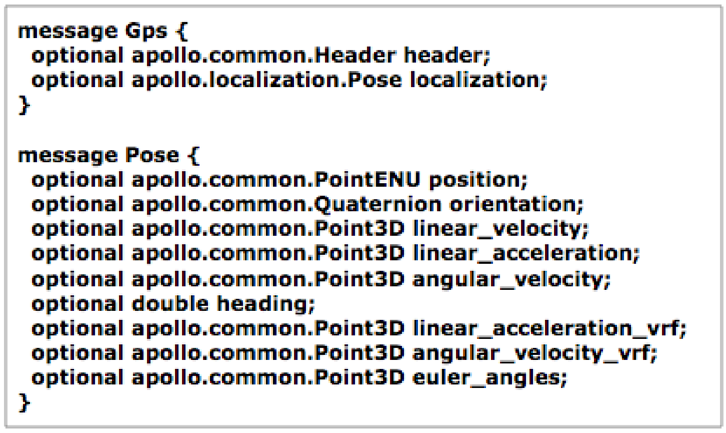
gps.localization().linear_velocity() is not accessed in TFBroadcaster => subscriber-side overprivilege

Figure 4.2: Concrete examples of publisher- and subscriber-side overprivileges.

level permission granting is actually not fine-grained enough to satisfy the least-privilege principle [185]. Based on our investigation on the Apollo AV software system, we discover overprivilege problems when messages are both published and subscribed to.

1. **Publisher-side overprivilege:** In message publishers, the values of some fields in published messages may be directly copied from messages that module subscribes to. In other words, the publisher only copies these fields without changing, but the granted publish permission, which allows both value copying and changing, permits more than what is actually needed.
2. **Subscriber-side overprivilege:** In message subscribers, certain fields in a message may be received but not used in the subscribing module. In other words, the subscriber is over granted with the subscribe permission for these fields.

Figure 4.2 illustrates a real example of subscriber-side overprivilege on the Gps message (defined in Figure 4.3) and publisher-side overprivilege on tf message [96] at the TFBroadcaster node of the GNSS driver module of Apollo. First, the localization.linear_velocity field in subscribed Gps messages is never used in any code path of TFBroadcaster node. Thus, this is a subscriber-side overprivilege on Gps.localization.linear_velocity because read permission for this field is granted to TF-



```
message Gps {
  optional apollo.common.Header header;
  optional apollo.localization.Pose localization;
}

message Pose {
  optional apollo.common.PointENU position;
  optional apollo.common.Quaternion orientation;
  optional apollo.common.Point3D linear_velocity;
  optional apollo.common.Point3D linear_acceleration;
  optional apollo.common.Point3D angular_velocity;
  optional double heading;
  optional apollo.common.Point3D linear_acceleration_vrf;
  optional apollo.common.Point3D angular_velocity_vrf;
  optional apollo.common.Point3D euler_angles;
}
```

Figure 4.3: Field definition of the `Gps` message in Baidu Apollo

Broadcaster but never used in it. Second, the state of the transform field in published tf messages is always copied from the localization field in subscribed Gps messages, which is a publisher-side overprivilege on tf.transform because TFBroadcaster does not need to change the value of tf.transform.

Even though these two example overprivilege problems are subtle, we find that they can have severe security and safety implications. As detailed in §4.8.1, an attacker can cause an AV running Apollo to lose sight of a front vehicle and crash into it by exploiting these over-granted privileges. This attack is demonstrated in our attack demo videos, and its validity has been confirmed by the Baidu Apollo developer team.

General existence of publish-subscribe overprivilege in ROS-based AV systems. Beside Baidu Apollo, we studied another popular open-source AV system Autoware [18], which is also built upon the ROS middleware. As shown in Table 4.1, we are also able to discover many publisher and subscriber-side overprivilege instances in a variety of key AV modules such as Perception, Planning and Actuation [18]. These results concretely show that the publish-subscribe overprivilege problem generally exists in ROS-based AV systems today.

Considering the general existence of the publish-subscribe overprivilege problem and its severity in AV systems, it is thus highly necessary to develop solutions to fully eliminate

Overprivileged node	Type	Affected topic	Affected fields
waypoint_follower's twist_gate [22]	Pub	/vehicle_cmd	ctrl_cmd, steer_cmd, accel_cmd, brake_cmd, gear, lamp_cmd, emergency
AS [19]	Pub	/as/arbitrated_speed_commands	speed
lidar_trackers obj_reproj [21]	Sub	/image_obj_tracked	total_num, real_data, lifespan
autoware_connector's can_odometry [20]	Sub	/vehicle_status	drivemode, steeringmode, gearshift, drivepedal, brakepedal, lamp, light

Table 4.1: Summary of overprivileged instances in Autoware. In the ”Type” column, ”Pub” means publisher-side overprivilege and ”Sub” means subscriber-side overprivilege.

the problem early at the AV system development stage. Thus, in this work, we fulfill this very need by being the first to develop a systematic solution.

4.2.3 Uniqueness of Publish-Subscribe Overprivilege in AV

Previous work considers overprivilege with publish-subscribe messaging [199, 120, 117, 194] at the coarser topic granularity, our work performs overprivilege detection and prevention at the message field granularity. Compared to previously-observed overprivilege problems in smartphone and smart home systems [142, 141, 143, 155], the publisher-subscribe overprivilege is unique in two aspects, creating both new design challenges and new opportunities for a practical solution. First, previous overprivilege problems occur in systems with regular user interactions like smartphone and smart home systems, where it is reasonable to rely on user judgment based on context to block unnecessary permission granting [200, 181, 155]. However, in the AV context, this is no longer acceptable since the whole design purpose is to enable autonomous driving without human input. In §4.5 and §4.6, we detail how we address this new challenge using static program analysis techniques.

Second, different from previous work on overprivilege in API accesses, the overprivilege here occurs when accessing message fields during the publish-subscribe communication of AV systems. This thus makes it possible to enforce access control policy entirely in the messaging layer and create a module-transparent solution. In §4.6, we detail how this domain-specific solution is created.

4.3 Threat Model

In this work, we assume that the attacker can fully compromise a single ROS module N in the victim AV system, which enables the attacker to: 1) sniff contents of arbitrary subscribed ROS-layer messages that N has been authorized to subscribe; 2) modify or inject ROS-layer messages that N has been authorized to publish; and 3) bypass or invalidate defense mechanisms implemented on N. Assuming a compromised ROS module is a typical threat model considered by previous ROS security work [199, 136, 122]. Though it is possible to compromise multiple modules, we believe that is harder for attackers and thus less realistic. For AV systems, such threat model is particularly realistic when M is a driver model for peripheral devices. Previous work has concretely demonstrated that all types of peripheral devices of an automobile, e.g., bluetooth and cellular, can be fully compromised remotely through common software vulnerabilities such as buffer overflow [125, 144, 171, 44, 36, 63]. By inspecting the commit logs of Baidu Apollo's open-source software repository [14], various patches can be found related to common implementation mistakes, such as out-of-bound array indices, uninitialized variables, and wrong definition of if-else conditions, in the driver modules of LiDAR, GNSS, radar and CAN bus devices [9, 10, 13, 7, 12, 8, 11, 5, 4, 6]. Thus, it is highly likely that similar software security problems in traditional automobiles' peripheral devices also exist in the driver modules of AV's peripheral devices, making these driver modules vulnerable to remote compromises.

We also assume that the underlying middleware (e.g., ROS) has been safeguarded with state-of-the-art authentication and access control mechanisms provided by Secure ROS (SROS) [199, 77, 85]. We assume that the access control policies, i.e., granted publish and subscribe permissions, are correctly enforced in SROS. Since this work focuses on the overprivilege problem described in §4.2.2, the security of the policy enforcement in SROS is out of the scope of this work.

Thus, after compromising module N, the attacker can perform any action allowed by the granted publish and subscribe permissions in SROS (introduced in §4.2). For example, the attacker can abuse the over-granted write permission of publish-overprivileged fields by publishing malicious information in the overprivileged fields, or abuse the over-granted read permission of subscribe-overprivileged fields by passively sniffing sensitive information from the overprivileged fields.

4.4 System Design

Figure 4.4 summarizes the design of our proposed fine-grained control of publish-subscribe permissions in *AVGuardian*. AVGuardian takes as input the source code of AV modules, the message format, and the current message-level publish-subscribe specification, that is, the messages each module registers to publish and subscribe. The fine-grained permission protection in AVGuardian has two major steps:

(1) Offline overprivilege detection: During AV software development, AVGuardian performs static program analysis to automatically examine each AV module's source code (from trusted AV developers) and detect overprivilege instances in the message fields of the messages defined by the module's publish-subscribe messaging specification. To prevent false positives in this detection which will lead to under-granting permissions in the online policy enforcement later, our static analysis tool is designed to handle complex real-world code with virtual functions and asynchronous callbacks.

(2) Online fine-grained access control: At run time once the software is deployed, for each detected publisher- or subscriber-side overprivilege instance, a fine-grained access control policy is generated and applied to the detected overprivileged message fields by online monitoring and policy enforcement. The access control is performed at the domain-specific publish-subscribe messaging layer so it is *module-transparent*, meaning that applying it does not require any changes to the AV module development process.

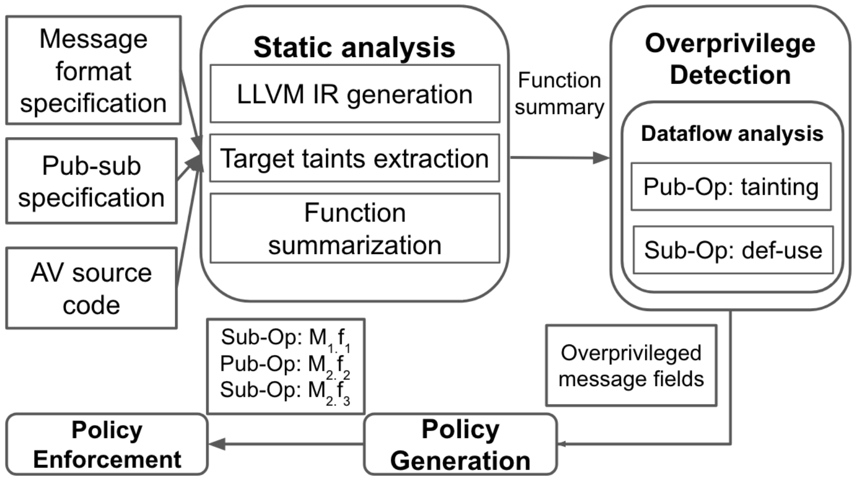


Figure 4.4: AVGuardian overview showing our publish-subscribe overprivilege detection and mitigation workflow. ”Pub-Op” stands for a policy on a message field with publisher-side overprivilege, ”Sub-Op” stands for a policy on a message field with subscriber-side overprivilege.

4.5 Overprivilege Detection Tool

Identifying both publisher- and subscriber-side overprivilege instances at the message field granularity is a prerequisite for policy generation and runtime enforcement to mitigate overprivilege in AV software. We propose to leverage static analysis to systematically detect overprivileged message fields on the publisher and subscriber side. Specifically, we design a static analysis tool for tracking flow-sensitive, field-sensitive and inter-procedural data flow (§4.5.2). Static analysis by design can achieve full code coverage of target software modules, which thus has the capability to achieve zero false negatives in data flow analysis and thus zero false positive in our overprivilege detection. Moreover, static analysis can pinpoint locations in the source code so that vulnerabilities can be proactively patched before deployment. Making no assumption on overlay AV modules, our tool can detect overprivilege in various ROS-based AV systems and is also extensible to detect other AV software vulnerabilities if they can be defined at the control/data flow level.

Besides the **field/object-sensitivity** requirement, to prevent removing true read/write permission for legitimate functionalities of an AV module, our tool need achieve *zero* false positive in the overprivilege detection. However, the extensive use of **virtual function** and **asynchronous event callback** in the complex C++ code base for AV systems [14, 18] can cause under-approximation in data flow analysis and lead to false positives of our overprivilege detection. In §4.5.3, we propose practical solutions to address these challenges to meet above two design requirements.

4.5.1 Static Analysis Overview

Pre-processing: We perform static analysis on function-level control flow graphs (CFGs) that are generated by LLVM intermediate representation (IR) of an AV module's source code. Analysis sources and sinks are determined based on the lifecycle and event callbacks of the module. We combine CFGs of callee functions that can be invoked along some control flow path from an analysis source to build an inter-procedural CFG (ICFG) for supporting inter-procedural analysis.

Subscriber-side overprivilege: We formulate the subscriber-side overprivilege problem as follows: within a module N, a field f in N's subscribed message M_s is over-granted with read permission if $M_s.f$ is never used for computation in any possible control flow path within N. We use inter-procedural define-use analysis at flow and field sensitivity to detect such instances (detailed in §4.5.2).

Publisher-side overprivilege: We formulate the publisher-side overprivilege problem as follows: within a module N, a field f in N's published message M_p is over-granted with write/modify permission if $M_p.f$ at publishing is never modified in any possible control flow paths within N, but is directly copied from certain fields in subscribed messages by N. Inter-procedural taint tracking at flow and field sensitivity is used for detecting such instances (detailed in §4.5.2).

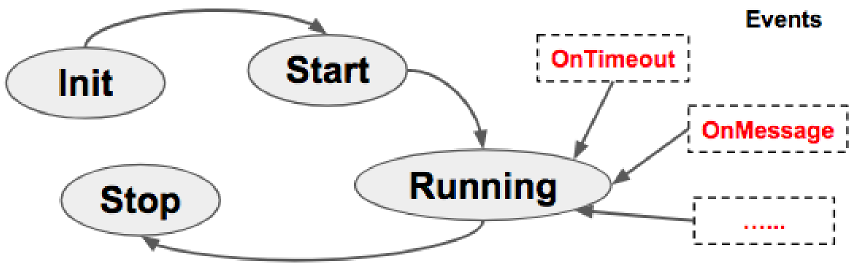


Figure 4.5: Lifecycle and event callbacks in an AV module

4.5.2 Dataflow Analysis Framework

AV module lifecycle: As illustrated in Figure 4.5, implementation of an AV module typically follows a predefined life cycle and usually includes multiple event callbacks for performing message subscription or periodic task processing (e.g., message publishing). In Apollo, when a module is launched, it first enters *Init* phase and then transits to *Start* phase, where key program states (e.g., publish-subscribe messaging interface) are initialized, message subscription event or periodic timer callbacks are registered and some ”main” entry function is called to start actual processing. When a module is stopped, it enters the *Stop* phase to clear its program states and terminate. Based on this observation, to ensure completeness, our dataflow analysis captures all possible entry points of the execution to form analysis sources: 1) *Init* and *Start* lifecycle functions, and 2) event callback functions registered in the *Init* or *Start* function. Analysis sinks are the calling instruction of ROS's message publishing API for publisher-side overprivilege analysis or the sinks of an ICFG for subscriber-side overprivilege analysis.

Inter-procedural support: Starting from one of above entry points, an inter-procedural CFG (ICFG) is generated by expanding the CFG of that entry point function in a recursive manner: for each function invocation along any possible control flow path, a CFG of the callee functions is generated and attached to the callee invocation point. Our anlaysis performs data flow tracking on this ICFG and also jumps into the CFG of a callee function to update the dataflow information of function parameters and class variables when a function invocation is met. The inter-procedural analysis can be computationally expensive without function summarization [201]. To ensure analysis efficiency, we perform summarization of functions that invoke message variables based on the caller-callee order. The current summary of a function f consists of 3 parts: 1) a target message variable set consisting of message variables that are defined in f or passed into f, 2) variables defined in f and tainted by the target message variable set, 3) define-use statements for variables in 1) and 2). Once a function is summarized, the subsequent analysis will directly read the summary to update the summary of current caller function when encountering it again.

Message taint tracking: Message instances instantiated from subscription may propagate across functions and class objects in an AV module. One common practice is storing a subscribed message instance into some member variable of a class object that may be later accessed by other functions in a module. Also, a message instance can be passed as parameters into a callee function where its fields can be read or modified before publishing. As a result, to detect read/write of message fields with completeness, we need to track originally subscribed message instances and their tainted variables at a field level across all control flows in an ICFG. In the field-sensitive context, field access on a message variable is also tainted. Assignments to elements of recursive data structures (e.g., array, list) are taken as tainting the entire structure. To support inter-procedural tainting, we perform and summarize above tainting within the CFG of a callee function that can be invoked in some control flow and whose parameters are tainted, and get back to the caller function to update its summary if the return variable is also tainted in the callee function. We preform this tracking process recursively until no new taints are found. A direct-copy label is also assigned on each left-hand side tainted variable to indicate if it is a direct copy or field access of the right-hand side tainted source. We detect direct-copy relationship by tracking all types of load, store, cast, getelementptr instructions in LLVM that invoke a tainted source as its source operand.

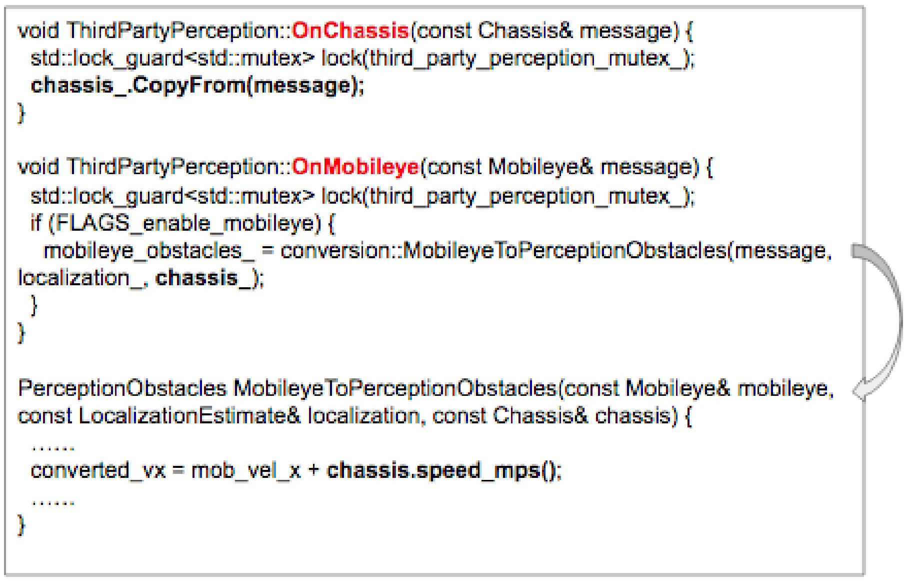
Overprivilege detection: To detect publisher-side overprivilege, we trace the taint tracking results on the target message variables: for all control flow paths from any entry point to a sink (i.e., message publishing), if a field f of the message variable M_p to be published is only tainted by some subscribed message instances with direct-copy label, $M_p.f$ is a publish-overprivileged field. To detect subscriber-side overprivilege, define-use analysis is performed on the target message variables and their tainted variables (with direct-copy label): along all control flow paths from any entry point to any sink of an ICFG, take the union of use statements on these variables to identify a set of accessed fields (i.e., fields with true read permission) and the other fields defined in a subscribed message are subscriber-side overprivileged by the current module.

4.5.3 Key Analysis Challenges

Field and object sensitivity: Detecting message field overprivilege requires tracking the data flow at field granularity. As shown in Figure 4.3, messages defined in the publish-subscribe message model in AV systems usually consist of many fields, some of which may be of composite or recursive types. Also, a message variable can be defined as a member in a class object. We support field sensitivity with the standard technique of detecting field access of a message variable based on `getelementptr` LLVM instruction and expanding it with an offset element inferred from the operands of `getelementptr`. Depending on the levels of composition in a composite typed field, the number of field-sensitive variables to be tainted and analyzed can become very large before primitive fields are hit. A configurable depth (3 by default) is defined to limit the level of field-sensitive analysis on a message variable. Also, we observe that some message fields at certain levels are not semantically meaningful to be differentiated further (e.g., the latitude and longitude value for a GPS field in a localization message). For recursive typed message fields (e.g., list, vector, map), a configurable length is also defined to limit the iteration on elements for analysis, which is a common practice in field-sensitive data flow analysis. This field-sensitive analysis also applies to class objects containing message variables.

Virtual function handling: Inheritance with base and derived classes in C++ are widely used in AV software to provide extensible interfaces for supporting multiple options of self-driving algorithms and vehicle models. However, this poses class binding uncertainty to our static analysis: at the LLVM IR level, calling a virtual function occurs through an indirect call and the target callee address is loaded through a variable determined at run time. To guarantee zero false positive in overprivilege detection, our analysis framework tracks data flows in virtual function calls with over-approximation by enumerating all possible derived classes. To identify all possible implementation of a virtual function defined in subclasses, we leverage LLVM's devirtualization pass to dump virtual table (`vtable`) entries and to get the index of each virtual function. Specifically, we detect instructions for loading a `vtable` pointer and determine which virtual function is invoked by the indirect call based on its type and the accessed `vtable` index. Then data flow analysis is performed in its implementation at each possible subclass.

Asynchronous event callbacks handling: The processing of asynchronous event callbacks and their ordering depend on runtime events, while at static analysis they are independent entry points. Assuming no or a specific order, however, may cause under-approximation in data flow tracking. As illustrated in Figure 4.6, assuming no or different orders of callbacks may lead to different data flow tracking results: if OnChassis callback is analyzed before OnMobileye, use on **speed_mps** field in the subscribed Chassis message is captured and otherwise missed. To avoid false positive in overprivilege detection, we enumerate all possible ordering among event callbacks in a module. To implement that, we define a synthetic entry point function containing an infinite loop within which all callback functions are added sequentially and the invocation of each one is predicated on some random condition. Figure 4.7 shows the resulted CFG for the synthetic entry point function that handles 6 asynchronous event callbacks, where arbitrary orders among callbacks are already encoded . Therefore, our data flow tracking need not take special handling on these



```
void ThirdPartyPerception::OnChassis(const Chassis& message) {
  std::lock_guard<std::mutex> lock(third_party_perception_mutex_);
  chassis_.CopyFrom(message);
}

void ThirdPartyPerception::OnMobileye(const Mobileye& message) {
  std::lock_guard<std::mutex> lock(third_party_perception_mutex_);
  if (FLAGS_enable_mobileye) {
    mobileye_obstacles_ = conversion::MobileyeToPerceptionObstacles(message,
localization_, chassis_);
  }
}

PerceptionObstacles MobileyeToPerceptionObstacles(const Mobileye& mobileye,
const LocalizationEstimate& localization, const Chassis& chassis) {
  ......
  converted_vx = mob_vel_x + chassis.speed_mps();
  ......
}
```

Figure 4.6: Order of message event callbacks affects data flow tracking.

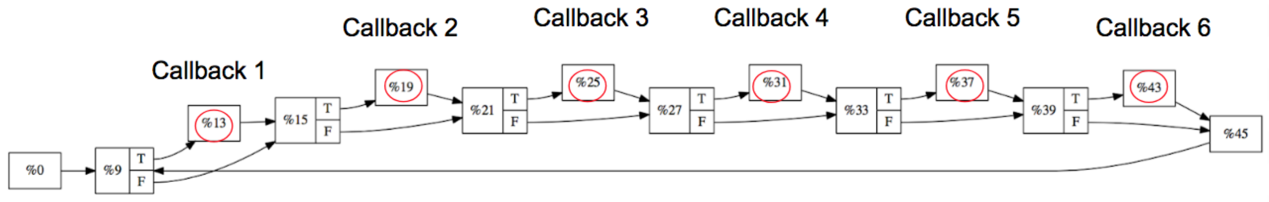


Figure 4.7: CFG for the synthetic entry point function, each block contains the definition of a callback. Dataflow analysis can be directly applied to these blocks.

event callbacks.

4.6 Overprivilege Mitigation

To prevent the publish-subscribe overprivilege problem, a straw-man solution is re-designing topic structures for the publish-subscribe model (e.g., chopping an existing topic into several sub-topics based on the read/write permission at message field granularity and having a module publishing or subscribing privileged sub-topics). However, this approach requires substantial changes in the current AV software, e.g., re-designing topics and how topics are handled, which hinders the deployability of such solution. Moreover, such more fine-grained topics significantly increase the number of messages and thus the messaging overhead at the run time, which makes it hard to meet the real-time requirement in AV systems.

In this section, we find that we can actually leverage the uniqueness of the overprivilege problem identified in this work to design a solution that is both module transparent, i.e., requiring no changes to the existing AV modules, and has low messaging overhead.

4.6.1 Access Control Policy Design

As shown in Figure 4.4, given the publisher- and subscriber-overprivilege instances detected by our static analysis, AVGuardian generates corresponding access control policies for each overprivilege instance during AV software deployment and applies low-overhead, module-transparent policy enforcement at runtime. Specifically, policies for subscriber-side overprivilege are defined based on the unused fields of a target subscriber, while policies for publisher-side overprivilege are defined based on the over-granted publisher, the fields that are over-granted write permission, and the source that was copied from. Violation of generated policies are detected online as anomaly indicators and pre-configured recovery strategies are performed. To highlight, our policy enforcement based overprivilege mitigation approach has the following key features:

- Online overprivilege policy violation detection and prevention
- Low performance overhead with acceptable delay to common ROS operations (e.g., module launch, publish-subscribe communication)
- Module transparency without requiring efforts from AV software developers or change of the AV software development process
- Flexibile run-time access control policy reconfiguration by simply changing the policy file for a module and restarting it

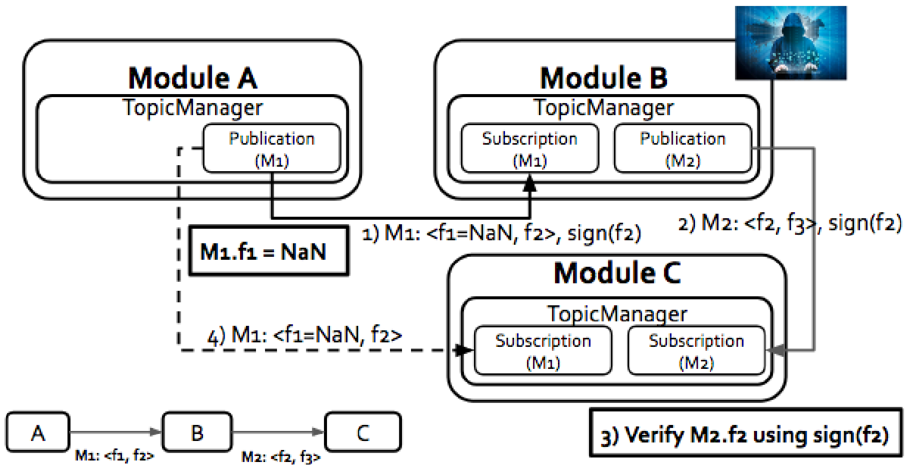


Figure 4.8: Detection and prevention of overprivilege policy violation. Assume module A publishes message M_1, module B (compromised by attacker) subscribes M_1 (with an overprivileged field f_1) and publishes M_2 (with an overprivileged field f_2 copied from $M_1.f_2$), module C subscribes M_2, and B is controlled by an attacker. Step 1-3 represent the message flow with policy enforcement on publisher- and subscriber-side overprivilege, where $sign(f_2)$ is the signature generated by Module A using $M_1.f_2$. Step 4 represents our proposed recovery strategy to directly contact publish originator when policy violation is detected.

4.6.2 Policy Enforcement

Our threat model assumes an attacker can compromise a single module to inject and run arbitrary code (i.e., code execution in other modules are legitimate and not compromised). Therefore, to ensure the effectiveness of our defense, AVGuardian needs to perform enforcement of generated access control policies on messages before they reach to a compromised module, since an attacker once compromising a module can arbitrarily bypass any defense logic or access control policies implemented on that module. Figure 4.8 shows an example where publisher- and subscriber-side overprivilege problems are identified on module B that is compromised by an attacker. Next, we use this example to help explain the following key functions in our proposed policy enforcement solution:

Subscriber-side overprivilege prevention: AVGuardian prevents subscriber-side overprivilege problems by proactively enforcing subscriber-side access control policies at publisher side (i.e., module A): given a message M to be published, for each subscriber S of

M, the publisher clears or sets meaningless values to fields in M that are unused at S. In this case, a module will receive no more information than it actually uses when subscribing to messages from other modules. Under this design, a message to be published needs to follow access control policies defined on a per-subscriber basis. Therefore, the performance overhead becomes proportional to the number of subscribers.

Publisher-side overprivilege policy monitoring: We use a digital signature-based solution for this function. First, from the static overprivilege detection, we can identify the original publisher (i.e., A) whose published message (M_1) is copied from by the publish-overprivileged module (B) to craft a published message (M_2), denoted as a *publish originator*. Then, by leveraging the key management feature of SROS [199], which has been integrated into ROS, a publish originator signs the source (i.e., $M_1.f_2$) of each overprivileged field before publishing a message. The publish-overprivileged module (B), if not compromised, will simply include an overprivileged field ($M_2.f_2$) without modification along with its signature into its published message. Finally, policy violations are checked at the subscriber (C) through verifying the signature of each overprivileged field to confirm that either the current value of an overprivileged field is consistent with that published by its publish originator or that a publish-overprivileged module has modified it. Using this method, AVGuardian can perform online detection if a publish-overprivileged module abuses the over-granted write permission at the message field granularity. If a policy violation is detected, AVGuardian reports this anomaly and activates recovery strategies pre-configured by AV developers.

Recovery strategies: When a publisher-side overprivilege policy violation is detected, we propose a solution, shown in Figure 4.8, that can recover the correct value of the overprivileged message fields with best effort and thus continue correct system operations. As shown, once the violation is detected, the subscriber (i.e., C) starts to subscribe M_1 from the publish originator (A) to obtain legitimate state for $M_2.f_2$ based on the latest $M_1.f_2$. Note that due to the asynchronous communication nature in publish-subscribe messaging, there is no guarantee that a copy of $M_2.f_2$ retrieved using $M_1.f_2$ is consistent to that from B. We make this design choice due to our observation that a module in AV software (e.g., Apollo) usually fetches the latest message in a subscription queue for its processing (e.g., `GetLatestObserved` calls of `AdapterManager` in Apollo).

Note that such best-effort recovery solution may not satisfy everyone due to the critical safety requirement of AV systems. For example, a conservative user may prefer to have an emergency stop for any anomly in the AV system. Thus, in AVGuardian, we provide a configurable interface to allow AV developers specify recovery actions based on their preferences.

Defense against replay attack: For publisher-side overpriviledge, even with the overpriviledged message fields signed by the publish originator, the attacker may still exploit the vulnerability using message replay attack, i.e., saving a signed message with values of its interest from the publish originator and replaying them later at a desired attack time. To defend against such attack, in our design we require a publish originator to sign the publish-overprivileged field with the publishing timestamp. The subscriber of a publish-overprivileged message maintains a message expiration window based on the one-way delay of an overprivileged field from its publish originator to its subscriber. When receiving an overprivileged message, it compares such window with the time difference between the current time and the publishing timestamp signed with the overprivileged field in the received message. Since it is typical that different modules in a AV system run in a single industrial PC [43], the modules share the same clock source and thus are already synchronized. If the time difference exceeds the expiration time window, potential replay attacks may be ongoing and the recovery strategies above can be applied accordingly.

AV developers can configure the message expiration window by profiling one-way delays from a publish originator to subscribers for each publish-overprivileged field. In our experiments, we choose 95-percentile of our profiled delays as a threshold. While false negatives or positives in replay attack detection may occur given the variation of message transmission, queuing, and processing delay, our empirical study in §4.7.3 using real-world AV system traces shows that for realistic exploitation scenarios, this mechanism can effectively detect replayed messages with both *zero* false positive rate and *zero* false negative rate.

Module transparency: We implement the policy enforcement in the ROS middleware between the AV software and commodity OS, by extending its *ros_comm* module responsible for communication among ROS nodes. Specifically, we implement our proposed solution in the `TopicManager` class that is defined as a singleton for each ROS node. Overprivilege access control policy files are read by the `TopicManager` during launch of a ROS node. The policy enforcement is performed by the `TopicManager` when it receives a message from its upper node to get published or receives a message from a subscription channel to be forwarded to its upper node. This such makes the policy enforcement process transparent to AV software, requiring no additional efforts from AV developers.

4.7 Evaluation

AVGuardian's overprivilege detection tool is implemented based on LLVM 3.4 and its runtime policy enforcement is prototyped through instrumentation of the ROS middleware (ROS Indigo used by Baidu Apollo [28]). We choose Apollo for our evaluation study because it is a popular production-level AV software platform with rapid growth of users and partners [15, 26, 62, 97, 89, 105, 104, 23]. We perform runtime profiling of AV modules in Apollo using real-world and fuzzed message traces to evaluate false positives/negatives in our overprivilege detection, and also micro-benchmarking and end-to-end evaluation on performance overhead of runtime policy enforcement.

4.7.1 Setup

Table 4.2 shows statistics of modules in Apollo 3.0 code base that we evaluated for overprivilege detection and the scale of fuzzed message traces used in runtime profiling for validation.

Module	# Publish topics	# Subscribe topics	# Functions	# Unique field input (# real value + # fuzzed value)
GNSS	13	2	6827	3100+7K
Velodyne	3	3	4773	77+10K
Pandora	2	2	4416	NA
Lslidar	5	3	4615	NA
RS-LiDAR	1	2	5317	NA
Canbus	2	2	9038	NA
Radar	2	0	3782	NA
USB Camera	2	0	2432	NA
Localization	2	2	15982	5415+7K
Routing	1	1	4831	2+11K
3rd-party perception	1	5	2868	233+11K
Perception	3	8	42485	NA
Prediction	1	4	10666	208+11K
RelativeMap	1	4	2955	560+11K
Planning	2	7	24397	420+10K
Control	1	4	8239	2300+10K
Monitor	3	15	4262	2+11K
Dreamview	8	19	12344	310+11K
Guardian	1	3	1392	2250+10K

Table 4.2: Statistics of evaluated modules in Apollo. At last column, ”NA” indicates that a module was not run due to lack of sensor stream input or GPU support.

Runtime message profiling:. We validate the overprivilege detection results through runtime profiling, by injecting realistic messages subscribed by a target module and capturing their usage events as well as messages published at run time, based on which we can validate that: 1) detected used fields in subscribed messages are indeed used at runtime; 2) detected unmodified fields in published messages remains unmodified across large number of input. To intercept such events, we instrument the protobuf library [73] (complemented by module instrumentation) to record use of message fields for our subscriber-side overprivilege validation. In addition, we record publish-subscribe messages on the fly using the `rosbag` utility provided by ROS and recover the in-and-out mapping of publish-subscribe messages at a target module based on their record timestamps.

Fuzzing input messages: We profile each module in Apollo using 3 demo input message traces (ROS bags) provided by Apollo that were captured from their real-world vehicle testing. To improve the confidence of our trace validation, we generate more diverse input traces for profiling by randomly fuzzing values of fields of input messages and using the demo input messages as our seeds. As shown in Table 4.2, more than 10K different values for each field are generated.

4.7.2 Accuracy of Overprivilege Detection

We validate our static detection results using results from runtime message profiling with real and fuzzed message input, and compute the false positives (FPs) and false negatives (FNs) to evaluate the accuracy of our tool. From the dataflow analysis perspective, a FP of subscriber-side privilege occurs when the static analysis determines a message field is unused, but at run time the field is used. A FP for publisher-side overprivilege occurs when static analysis determines a field is unmodified by a module, but at run time the field is written. For both subscriber-side and publisher-side overprivilege, an observed FN occurs when static analysis detects a field as used or modified but no reads or writes are observed at run time.

As shown in Table 4.3 and 4.4, comparing with the profiling results, we observe zero false positives in overprivilege detection of both types, which validates that our dataflow analysis should cover all possible execution paths and by design achieve zero false positive in overprivilege detection. Our data-flow analysis may over-approximate used/modified message fields due to conservative resolution of virtual functions and asynchronous event callbacks, leading to FNs of overprivilege detection. However, we only observe 3 false negative of subscriber-side overprivilege and 1 false negative for publisher-side (last row of Table 4.3 and 4.4). Note that these FNs cause no functional errors in policy enforcement (i.e., not blocking true read/write permission). Such low FN rate (less than 1.8%) also demonstrates the effectiveness of our overprivilege detection in attack surface reduction. The subscriber-side FN is due to limitations in the number of inputs available to exercise code paths at runtime, whereas the publisher-side FNs are caused by loading static values from configuration files into messages.

We also break down the accuracy improvement with analysis enhancements to address virtual function and asynchronous callback issues. The virtual function enhancement significantly reduces FNs in define-use analysis (or equivalently FPs in subscriber-side overprivilege detection) and FNs in detecting direct copy from subscribed to published message fields (publisher-side overprivilege), by tracking data flow in more control flow paths. The event callback order enumeration targeting at specific modules further improves the precision of the overprivilege detection.

Tool w/o features	**Detected #**	**FP #**	**Observed FN #**	**Observed FP #**
w/o virtual function	551	28	NA	NA
w/o callback ordering	525	2	NA	NA
w/ both	523	0 (baseline)	3	0

Table 4.3: Evaluation of accumulative improvement of subscriber-side overprivilege detection. In total 523 TPs.

Tool w/o features	**Detected #**	**FN #**	**Observed FN #**	**Observed FP #**
w/o virtual function	50	6	NA	NA
w/o callback ordering	55	1	NA	NA
w/ both	56	0 (baseline)	1	0

Table 4.4: Evaluation of accumulative improvement of publisher-side overprivilege detection. In total 56 TPs.

Running time. As a part of the offline analysis, our tool summarizes functions of a module in Baidu Apollo in roughly 15 minutes to several hours depending on the code size. The computed summary can be reused later for overprivilege detection. After the summarization, the overprivilege detection process completes in less than a minute. Note that the overprivilege analysis is an offline task and thus does not affect the runtime system performance. Also, the efficiency can be further improved by analyzing functions in parallel [202].

4.7.3 Effectiveness of policy enforcement

We evaluate the performance overhead due to runtime policy enforcement in an Apollo docker container using real input messages provided by Apollo. A 1024-bit RSA key is generated for signing publisher-side overprivileged fields using a popular cryptography library libgcrypt [58]. As our container runs the same Apollo codebase installable on real vehicles, the only contributing factor for performance difference on real vehicles would be hardware capabilities. Apollo's documentation recommends real vehicles to run Apollo in an industrial PC [43] with CPU configuration (Intel Xeon E3-1275v5 3.6GHz) more powerful than that of our server (Intel Xeon E5-4620v2 2.6GHz). Thus, the following performance results (e.g., message latency) should be better on real vehicles.

Overhead analysis. Our policy enforcement requires enforcing a policy on a message to be published towards a certain subscriber. Table 4.5 shows the overhead of extra operations for enforcing a policy for publisher and subscriber-side overprivilege. At module launch a module needs to read a list of policies and load its key to memory. The average launch time of an AV module without policy enforcement is around 1.58 seconds and increases by 1% with enforcement. During publish-subscribe communication, a publisher may enforce subscriber-side overprivilege policies by clearing unused fields of a message to each target subscriber, and publisher-side policies by signing the source of an overprivileged field. The subscriber may thus need to verify the signature. Given 10ms baseline latency for publish-subscribe on medium-size message and the overhead of each operation shown in Table 4.5, we propose to accelerate the digital signature generation by specialized hardware to improve the scalability of our policy enforcement design [191, 193, 153].

Source of overhead	**Overhead (ms)**
Policy parsing (100 policies)	0.8
Key loading	0.2
Per-field clearing	0.15
Per-field signing	3
Per-field verifying	0.15

Table 4.5: Performance overhead introduced by the policy enforcement design. Based on our overprivilege detection results, the number of policies on a module in reality is fewer than 100.

End-to-end overhead. We evaluate the end-to-end performance overhead of AV's decision

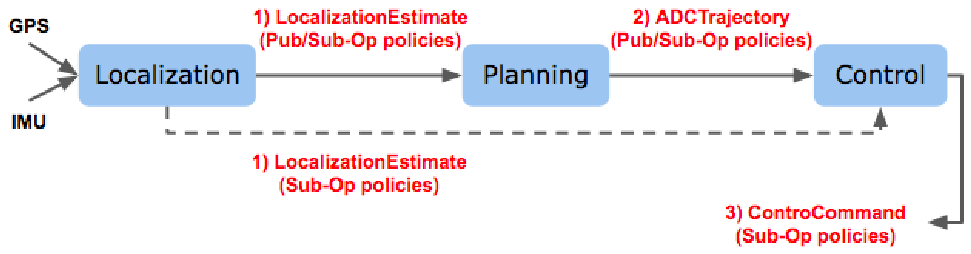

GPS
IMU
Localization
1) LocalizationEstimate (Pub/Sub-Op policies)
Planning
2) ADCTrajectory (Pub/Sub-Op policies)
Control
1) LocalizationEstimate (Sub-Op policies)
3) ControCommand (Sub-Op policies)

Figure 4.9: End-to-end policy enforcement evaluation setup

cycle shown in Figure 4.9: 1) Localization module subscribes GPS and IMU messages that are played out from collected trace and publishes LocalizationEstimate messages; 2) Planning module subscribes LocalizationEstimate, Routing and Chassis messages (also played out from trace) and publishes ADCTrajectory messages; 3) Control module subscribes ADCTrajectory messages and publishes a ControlCommand message. In particular, policies apply on one publish-overprivileged field in ADCTrajectory and ControlCommand message and several subscribe-overprivileged fields in LocalizationEstimate, Planning and ControlCommand messages. Policy enforcement is performed at message publishing by each of these three modules: each needs to clear subscribe-overprivileged fields before publishing to a target subscriber and Localization and Planning module as publish originators need to generate signatures for one publish-overprivilege field in ADCTrajectory and one in ControlCommand message.

To evaluate the end-to-end performance overhead, we measure the end-to-end delay from when a LocalizationEstimate message is to be published until a ControlCommand message is published by Control module to close current control cycle. The end-to-end delay without policy enforcement is on average 122ms. With enforcement of publisher and subscriber-side overprivilege polices, this end-to-end delay increases to 132ms. We validate that this 10ms overhead does not affect AV decision logic, since the message sequence and order were confirmed unchanged with and without our policy enforcement. Also, digital signature operations can be accelerated using GPUs (required by Apollo [43]) to significantly reduce this overhead (e.g., 12.7x speedup [153]).

Defense effectiveness against message replay: To evaluate our replay attack defense mechanism, we use real-world traces captured from Baidu Apollo's testing on local roads of Sunnyvale, California [112], and simulate message replay attacks by replaying the most recent publish-overprivileged field other than the up-to-date one always leads to an exploit. From the results, we analyze whether there exists an expiration window value that fully separates the replayed copy and the up-to-date copy based on the time gap from when the overprivileged field is sent by a publish originator to when it reaches a subscriber. Our analysis results show that in this worst case our defense mechanism can indeed have false positives and false negatives, but its Area Under the Curve (AUC) [140] is still greater than 0.99, which is generally considered acceptable [140].

Meanwhile, we further find that for specific exploitation scenarios, since the attacker usually needs messages with specific values in the overpriviledged fields to cause meaningful damages at the vehicle control level, our relay attack defense mechanism can actually achieve *zero false positive and false negative rates*. To evaluate this, we experiment our defense mechanism on two concrete exploitation scenarios we construct, TF attack and PCL attack, which can both cause vehicle crashes as detailed later in §4.8.1 and §4.8.2.

In this evaluation, we simulate message replay attacks by letting the malicious node capturing and replying messages with the attack-desired values, and still use the real-world traces captured by Baidu Apollo [112]. Specifically, for the TF attack scenario, we examine old GPS messages with exploited fields set using values that can cause obstacles to be relocated out of the AV's current traffic lane. Our results show that only messages at least 4 seconds before can cause the relocation, while the profiled one-way message delay from the GNSS parser nodelet to the perception module is less than 110 milliseconds. Thus, our message replay defense mechanism in the perception module can detect such attack without incurring any false positives or negatives. For the PCL attack, since PointCloud messages with zero height or width values may only happen when the LiDAR sensor is broken, we cannot observe any such messages in our real-world traces and thus there are no replay attack opportunities.

4.8 Findings

Our tool detects 523 subscriber-side overprivilege instances and 56 publisher-side overprivilege instances in the Apollo code base. Through investigation on the use of these overprivileged fields, we identify important security or privacy implications with 9 publish/subscribe-overprivileged instances in 11 ROS nodes, confirmed by code validation and attack demonstration. Table 4.6 summarizes details of these overprivilege instances and security findings. These overprivilege vulnerability instances exist in Apollo 1.5/2.0/2.5/3.0 (latest version at the time of our study) and as demonstrated by us can lead to two major types of security consequences: 1) vehicle collision caused by attacks manipulating perceived obstacles; 2) leakage of sensitive information, such as Vehicle Identification Number (VIN), GPS data, etc., that can lead to theft of AV owner's personal information, VIN cloning to hide the identity of stolen vehicles and real-time tracking of AV passengers.

To demonstrate the severe security impacts of publisher-side overprivilege in our threat model, we construct two attacks, called the TF and PCL attacks, which exploit publish-overprivileged fields in driver modules for the GNSS and LiDAR devices (assuming they are compromised) to disrupt the LiDAR processing in the perception module of Apollo, leading to relocation or removal of front-facing perceived obstacles. When evaluated using Apollo's built-in simulation environment (SimControl) and its provided real-world traffic traces, both attacks cause an AV to crash into an obstacle. Figure 4.10 shows the simulated attack outcomes of both attacks in SimControl. Moreover, to demonstrate the privacy implication of subscriber-side overprivilege vulnerability, we construct an attack called VIN stealing attack that exploits a subscribe-overprivileged field in the GNSS driver module (assuming it is compromised) to steal the Vehicle Identification Number (VIN)

Overprivileged node	Type	Affected topic	Affected fields	AV security implication
TFBroadcaster	Pub	tf	transform.translation	Relocate perceived obstacles to cause collision in TF attack (4.8.1)
TFBroadcaster	Pub	tf	transform.rotation	Reduce the perceived size of obstacles
Velodyne's compensator	Pub	PointCloud	width, height	Remove perceived obstacles to cause collision in PCL attack (4.8.2)
Pandora's, Lslidar's, RS-LiDAR's motion compensator	Pub	PointCloud	width, height	Remove perceived obstacles to cause collision
Control	Pub	ControlCommand	signal	Manipulate the on/off state of signal light
TFBroadcaster	Sub	GPS	pose.linear_velocity	Vehicle speed reconnaissance for launch TF attack (4.8.1)
GNSS	Sub	Chassis	license.vin	AV owner's identity theft in VIN stealing attack (4.8.3)
Perception, 3rd-party perception, RelativeMap, Control	Sub	Chassis	chassis_gps	AV's location privacy leakage
Prediction, Control	Sub	Planning	debug.routing	AV's route privacy leakage

Table 4.6: Summary of overprivileged instances with security implication. In the "Type" column, "Pub" means publisher-side overprivilege and "Sub" means subscriber-side overprivilege.

of an AV and highlight serious identity and privacy theft to AV owners. We performed responsible disclosure of our attacks to Apollo developer team and received acknowledgement confirming that our attacks are valid.

4.8.1 Obstacle Relocation Attack (TF Attack)

A TF attack exploits a publish-overprivileged field defined in the **/tf** message that is published by the TFBroadcaster nodelet in the GNSS driver module. TFBroadcaster [95] is a ROS nodelet that subscribes to Gps (i.e., /apollo/sensor/gnss/odometry) topic and publishes /tf message. /tf message is mainly used by the LiDAR and radar processing subnode in the perception module to construct pointcloud-to-world coordinate system that is used for estimating the position of perceived obstacles in the physical world. AVGuardian detects over-granting write permission of tf.transform to TFBroadcaster: TFBroadcaster does not compute tf.transform field, but copies the value of certain sub-fields of localization field in subscribed Gps messages to tf.transform.

Attack construction: We discover that the two subfields of tf.transform, **translation** and **rotation**, are used to perform the affine transformation to estimate the position and size of a perceived obstacle in the physical world, where A is constructed using rotation and b using translation.

$$y = Ax + b$$

Specifically, translation consists of values for the x, y, and z dimensions: adding an offset to certain dimension will relocate the estimated position by the same offset on that dimension. Therefore, by manipulating the translation field in a /tf message to be published, an attacker can cause the LiDAR subnode to perceive an obstacle that is moving ahead on the same lane to be on another lane and at a farther distance (illustrated in 4.11). In the worst case, surrounding obstacles on the road can be relocated out of the road and may be treated as background objects rather than obstacles. To make the attack worse, an attacker can further exploit the subscribe-overprivileged field localization.linear_velocity of the Gps message on TFBroadcaster (illustrated in Figure 4.2) to perform reconnaissance on the linear velocity of an attacked AV and launch a TF attack in high speed traveling situations (e.g., when localization.linear_velocity is higher than 60mph). Based on this observation, assuming that TFBroadcaster is controlled by an attacker, we construct an attack through abusing the write-permission of tf.transform field by adding no more than 15 to its x and y dimension values in a series (5-second duration) of /tf messages published from TFBroadcaster. The reason for sending multiple exploited messages is based on the observation that the prediction module in Apollo maintains 5-second memory for each perceived obstacle generated from the perception module: when an obstacle under tracking is invalidated by the output of perception module, it is still tracked as an obstacle by the prediction module for the subsequent 5 seconds and will still be considered for path planning. Therefore, to completely relocate an obstacle, exploited /tf messages need to be continuously sent for at least 5 seconds. Validated in SimControl with sensor traces captured from real-world testing provided by Apollo, the TF attack causes an AV running Apollo's self-driving software to perceive obstacles that are moving ahead on its same lane as if they were on another lane, and results in obstacle collision.

AVGuardian's defense solution. To prevent a TF attack following AVGuardian's mitigation approach, the publisher of Gps topic, namely the parser nodelet in the GNSS module, uses its private key to generate signatures for two relevant fields of a Gps message to be published, one for localization.position and one for localization.orientation, and appends them to the fields when publishing. In legitimate operations, TFBroadcaster copies Gps.localization.position and Gps.localization.orientation along with their signatures to the transform field of a /tf message to be published. Any subscriber of /tf topic, such as the perception module, will verify both sub-fields in tf.transform using the public key of the GNSS parser nodelet to detect any abuse of write permission on the tf.transform field by TFBroadcaster. If any signature cannot be verified, the subscriber will flag anomaly and trigger a recovery solution, for example, by requesting a latest Gps message from the GNSS parser.

4.8.2 Obstacle Removal Attack (PCL Attack)

The PCL attack exploits publish-overprivileged fields defined in **PointCloud** message that is published from the compensator nodelet in the Velodyne driver module. The Velodyne compensator [29] is a ROS nodelet that performs motion compensation for point cloud data collected by Velodyne LiDAR devices [101]. It subscribes to /apollo/sensor/velodyne64/PointCloud2 messages from the Velodyne converter nodelet and publishes PointCloud (/apollo/sensor/velodyne64/compensator/PointCloud2) topic. Both topics are defined as sensor_msgs/PointCloud2 type in ROS [79], which consists of several metadata fields and a data field for encoding attributes and actual contents of point cloud data, respectively. These published PointCloud messages serve as point cloud input to the LiDAR processing subnode in the perception module of Apollo to construct surrounding obstacles. AVGuardian discovers that write permission is over-granted to all fields beside data of PointCloud topic to the compensator node: only the data field is modified while the rest of fields (e.g., width, height) are field-wise copies of subscribed /apollo/sensor/velodyne64/PointCloud2 messages.

Attack construction:. The LiDAR processing subnode in Apollo's perception module pre-processes input PointCloud message (implemented in TransPointCloudToPCL) using pcl library of ROS, where a two-level loop is defined to perform element-wise copy of the bytes in the data field of a PointCloud message to construct point cloud input for obstacle identification (implemented in fromPCLPointCloud2 [37]). In particular, we observe that two metadata fields **width** and **height** in PointCloud message, defining the size of the point cloud data, is used in the boundary condition for the nested looping. If the value of width or height field is reduced, the LiDAR subnode will construct obstacles using incomplete point-cloud data. By setting either of the fields as 0, point cloud frames will not be taken into the perception surrounding obstacles can be zeroed out (illustrated in Figure 4.12). Based on this observation, assuming that the compensator node is controlled by an attacker, we construct an attack through abusing the write-permission of height field by setting it to 0 for 5 seconds at the compensator node. This time duration is also determined based on the 5-second memory for each tracked obstacle in the prediction module (explained in §4.8.1). Therefore, to completely zero out an existing tracked obstacle, exploited PointCloud messages need to be published continuously from the compensator node for at least 5 seconds. Validated in SimControl with the same sensor traces used in the TF attack, the PCL attack causes perceived front-facing obstacles moving on a same lane to temporarily disappear and results in collision with an obstacle.

AVGuardian's defense solution: To prevent a PCL attack following AVGuardian's mitigation approach, the publisher of /apollo/sensor/velodyne64/PointCloud2 topic, namely the converter nodelet in Velodyne driver module, uses its private key to generate one signature for height and another for the width field of published messages. Similar to defense against TF attacks, any subscriber of PointCloud topic (e.g., perception module) verifies the height and width field of a subscribed PointCloud message using the public key of Velodyne converter nodelet to detect any abuse of the write permission on height and width field by Velodyne compensator nodelet. If any signature cannot be verified, the subscriber should flag the anomaly and trigger a recovery solution, for example, by requesting a new /apollo/sensor/velodyne64/PoitCloud2 message from Velodyne converter nodelet.

PCL attack feasibility in other LiDAR drivers: We also apply our detection tool to other LiDAR driver modules that are available in Apollo 3.0 code base. Besides Velodyne driver, our tool also detected a same publish-overprivilege problem on the compensator node of the driver module for Pandora's LiDAR device [69], Lslidar and RS-LiDAR.

4.8.3 VIN Stealing Attack

Chassis message in Apollo 3.0 includes a field **license** that contains the Vehicle Identification Number (VIN) of an AV [25]. AVGuardian detects that this field is set in each Chassis message to be published by the Canbus module, but remains unused by any Chassis subscriber, including the GNSS driver module. As a result, an attacker controlling the GNSS driver module can passively sniff this subscribe-overprivileged field to steal the VIN of an attacked AV, using which to uncover personal information of the AV owner, including name, address, and in some cases phone number and email address [108]. Furthermore, attackers can use a single stolen VIN to register dozens of stolen vehicles for masking vehicle theft or filing insurance claims on totaled vehicles, and even to make duplicate keys for an attacked AV [30]. Using AVGuardian's mitigation approach in §4.6, such attack can be fully prevented by enforcing a corresponding subscribe-overprivilege policy at runtime to set the license field of each Chassis message published from the Canbus module to zero.

4.8.4 Apollo Developer Feedback

We performed responsible disclosure and received confirmation from the Apollo developer team that our attack findings are valid under our threat model. We also obtained valuable insights to the cause of such overprivilege problems in AV software. Based on feedback from the Apollo developer team, the cause of overprivilege is likely due to the fact that the Apollo team aims to encourage open-source contribution to Apollo and thus provides a unified and liberal message interface. They commented that it can be highly beneficial to have a systematic approach to automatically uncover and prevent overprivilege problems, which is exactly the research goal of this work.

4.8.5 Publish-Subscribe Overprivilege Fixing

Based on our feedback from Baidu Apollo developers, the major reason for the existence of publish-subscribe overprivilege across old versions to the latest one, is that they provide a sparse set of fields for a message topic in order to allow open contribution of new functionalities leveraging the available fields in a message. Also, due to the multidisciplinary development nature of AV software, it is hard to enforce unified security design principles and polices across a large team of developers with different domains of expertise. Given that AV software is at an early, active development stage, it is hard to expect what fields will be used as the development goes from an AV system designer's perspective. One feasible fixing solution of this overprivielge problem is redesigning topic structures for the publish-subscribe model (e.g., chopping an existing topic into several sub-topics based on the read/write permission at message field granularity and having a module publishing or subscribing privileged sub-topics). However, this approach requires substantial and constant changes in the current AV software, e.g., re-designing topics and how topics are handled across different AV modules. Such more fine-grained topics significantly increase the number of messages and thus the messaging overhead in the run time and poses another challenge to satisfy the real-time requirement in AV systems. In Apollo code base, we observe that 24 new message topics (40% increase on the total number of topics) need to be created using this approach and require changes of the message handling logic in 11 modules.

4.9 Limitation & Scope

False negatives from overprivilege detection: Our current tool performs data flow analysis on only target message variable and their propagated taints, and taint propagation only considers certain types of LLVM instructions. Thus, our tool may miss implicit pass-through in our publisher-side overprivilege detection (e.g., adding a target variable with a constant or a constant variable will not propagate the taint) and result in FNs in publisher-side overprivilege detection. We leave the resolution of this problem as future work since (1) these cases are not prevalent in real-world AV code bases, e.g., we only observed one in our evaluation (§4.7), and (2) FNs will not affect the functional correctness of the system since they does not lead to under-granting of permissions in our policy enforcement.

Potential scalability issue in policy enforcement: AVGuardian's policy violation detection on publisher-side overprivilege requires signing each overprivileged fields and the cost is proportional to the number of overprivileged fields existing in a message, which may cause scalability issue to our solution. As future work, we plan to address this issue by integrating crypto acceleration approaches [191, 193, 153] to AVGuardian. Also, our proposed recovery solution against violation of publisher-side overprivilege policy, through contacting the publish originator, only provides best-effort recovery of the valid state of a publisher-side overprivileged field. However, similar to our recovery solution, we observe that it is common to use the latest state of a message field for computation in different AV modules.

Threat model scope: AVGuardian targets at defending against attacks exploiting the overprivilege at the publish-subscribe communication model, and thus does not handle other attacks, such as spoofing of non-overprivileged fields to the publish-subscribe communication channel at publisher side. A publisher, even if not compromised, must be granted write permission on those fields and AVGuardian cannot differentiate valid or spoofed states for them. Also, we do not consider scenarios that a compromised module maliciously withholds messages to be published.

4.10 Summary

In this work, we design and implement AVGuardian that systematically detects message field level overprivilege for the publish-subscribe communication in AV systems and performs policy enforcement on overprivileged fields at runtime to mitigate their security damage. AVGuardian performs flow- and field-sensitive data flow analysis with enhancements on virtual function and asynchronous callback handling to achieve high precision in overprivilege detection and prevents under-granting necessary publish/subscribe permissions. Its policy enforcement is transparent to AV modules and incurs low performance overhead to publish-subscribe communication. AVGuardian discovers 579 overprivilege instances in Baidu Apollo, which lead to concrete exploits causing vehicle collision and identity theft for AV owners that have been confirmed valid by the Apollo developer team.

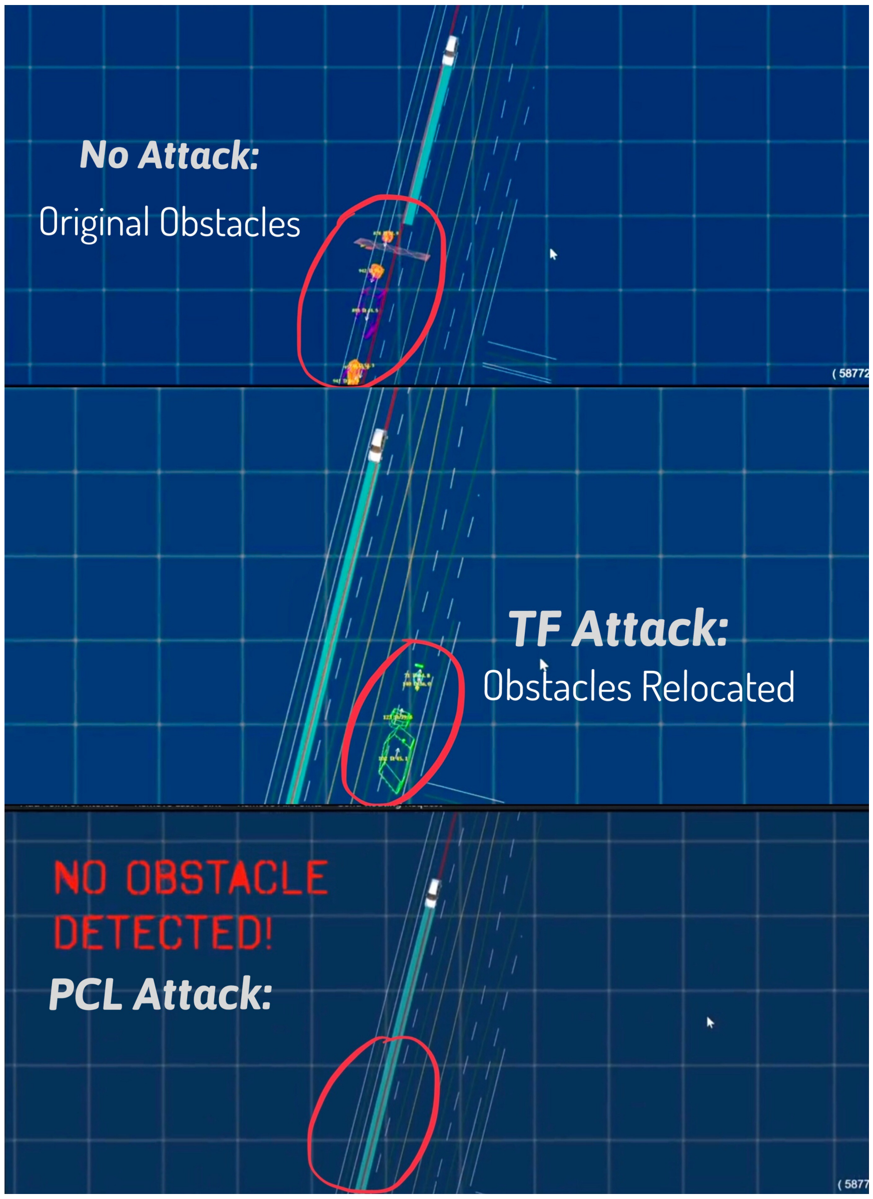


Figure 4.10: Comparison of attack outcomes in a same road and traffic scenario: obstacle relocation by TF attack vs. obstacle remove by PCL attack

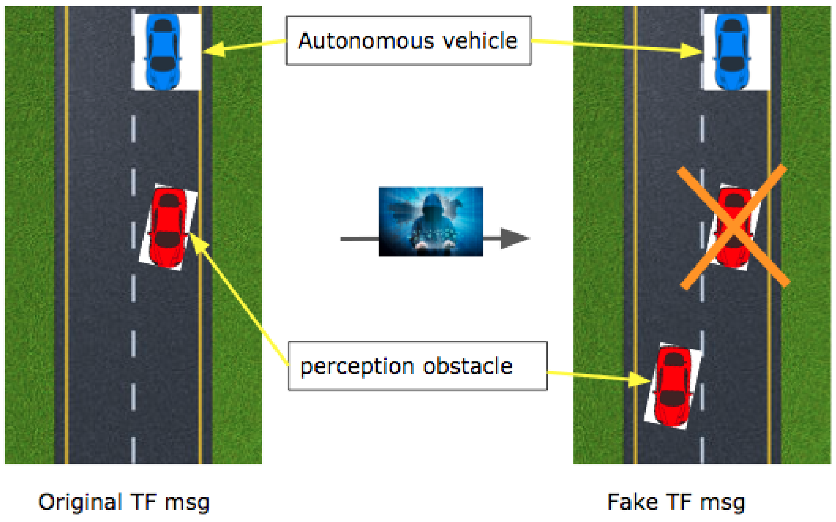


Figure 4.11: Relocating obstacles in AV's perception view through exploiting publish-overprvilege of /tf message on TFBroadcaster

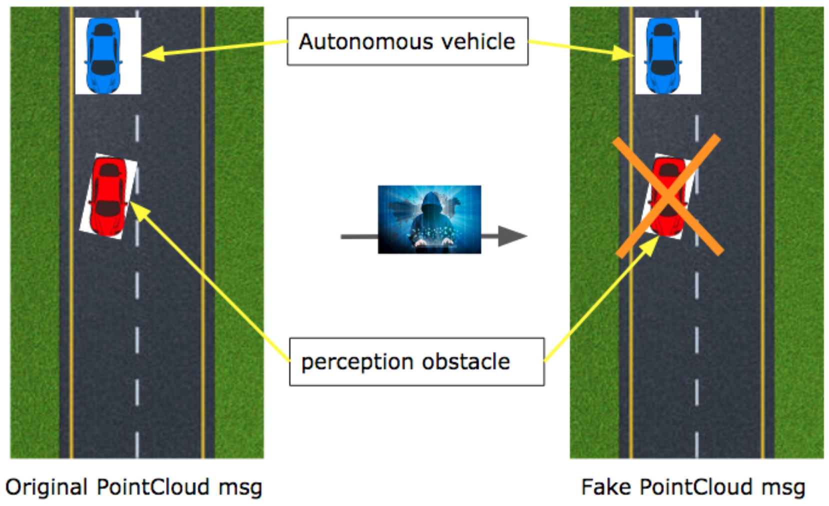


Figure 4.12: Removing obstacles from AV's perception view through exploiting publish-overprvilege of PointCloud message on Velodyne compensator

CHAPTER V

AVerifier: Verifying Driving Safety Compliance for Autonomous Vehicle Software

5.1 Introduction

Emerging Level 4 and 5 fully autonomous vehicles (AV) hold great promise in transforming today's transportation systems and mobility services. However, a recent survey shows that 50% of consumers have safety concerns on AVs [90]. Unanimously agreed by AV software vendors and the government authorities, driving safety, in particular verified compliance with well-recognized safety standards (e.g., traffic laws, voluntary safety standards for AVs from NTHSA [17], Responsibility-Sensitive Safety [186]), is the most important design requirement for AVs before their deployment in the real world at large scale. However, there is a lack of systematic approach to sufficiently verify the safety compliance of the design and implementation of AV software with certain guarantees.

Existing AV safety testing approaches (e.g., simulation, close track testing, on-road testing) mainly fall in the black-box dynamic testing category. They take test case input generated from real-world traffic scenarios and validate the correctness of AV software by comparing its output with the expected outcome. As a result, the effectiveness of existing approaches highly depends on the quality and diversity of the test cases. However, due to the infancy of autonomous driving technologies at the current stage, regulation barriers and high cost/risk of real-world testing, it is unrealistic for AV vendors to collect a sufficiently diverse set of traffic scenarios for test case generation. Corner cases for various traffic scenarios are widely thought as a major paint point for the current AV safety testing and validation. To address this inefficacy, we propose AVerifier, a static program analysis framework that verifies the compliance with standard safety policies in the code implementation of AV software in a more systematic manner. AVerifier aims to support the verification of a diverse set of safety policies for AVs, including the AV safety elements (e.g., operational design domains, object and event detection and response, minimal risk condition, etc.) defined by U.S. Department of Transportation [17], existing traffic laws, Responsibility-Sensitive Safety [186], etc. We develop our approach towards providing strong guarantees of zero false negatives in compliance verification: once compliance of a rule is verified, zero violation of that rule is guaranteed in the software logic. Our approach differs fundamentally from existing black-box AV testing solutions. Leveraging rigorous programming language and software engineering techniques, AVerifier provides completeness guarantee in code verification. Moreover, its white-box code analysis provides easy-to-reason hints for debugging and policy enforcement. In addition, its systematic test case generation capability enables reproducing identified flaws under realistic traffic scenarios. Also, our approach can provide other new advantages including improving testing efficiency and reducing test cost for AVs.

AVerifier takes the source code of AV software and safety policies/rules as input, transforms the safety rules into some specification format that can be directly analyzed at the code level of AV software based on an AV domain-specific semantic mapping and presents to AV developers any potential inconsistency between the specification and source code detected from the static analysis. Symbolic execution can be further applied to the analyzed code paths to systematically explore the path feasibility and generate any violation-triggering test cases to AV developers for further validation. Two key challenges need to be addressed in order to achieve the systematic and complete verification guarantees in AVerifier. First, AV safety rules (e.g., traffic rules)are usually very domain specific. A semantic gap exists between policy specifications and the AV code-level behaviors. Thus, an expressive specification interface is required to allow AV software developers to define high-level safety policies and then handle their mappings to code-level patterns. However, how to develop such interface is non-trivial since it needs to be both rich in AV semantics and also directly expressible at the AV code level. To incorporate unique higher-level semantics involving complex decision/control logic into the static analysis process in our verification, AVerifier generates AV-specific safety specification with rich road traffic and driving semantics (e.g., vehicle/traffic states) through an inferred mapping between the high-level AV semantics and code-level implementation. Second, the extensive use of object-oriented programming in AV software development increases the complexity of the software logic and poses engineering challenges to control dependence analysis for identifying code relevant to safety rule implementation. To address this challenge, we consider beyond control dependence and leverage program dependency analysis to identify more code pieces relevant to the safety rule implementation in AV software. Validation on safety rules defined in existing traffic laws using Baidu Apollo's APIs demonstrate the generality of the specification interface, as all implemented rules can be expressed into our policy specification form. Also, our initial evaluation results on Baidu Apollo indicate that the current AVerifier's analysis approach captures the code-level implementation of common safety rules and uncovers some noncompliance with the safety requirement defined in existing safety standards (e.g., instructions for driver license tests).

The contributions of this work are as follows:

- To the best of our knowledge, we are the first to propose a practical solution for verifying the traffic semantic-level safety requirements of AV software with completeness guarantees by design.
- We design and implement a program dependence analysis framework to help AV developers perform static detection of safety requirement noncompliance problems

in AV software. Based on the dependence analysis output, we propose to further use symbolic execution in a scalable manner to generate violation-triggering test input in a systematic manner. Our analysis uncovers the noncompliance with some standardized safety requirements in an early version of Baidu Apollo.

- We propose a safety policy/rule specification interface by constructing a unique abstraction of the rich road traffic and driving semantics and a semantic mapping that relates each semantic entity to its code-level implementation in AV software. Through this interface, AV developers are able to express common safety rules in a composable and flexible manner. We validate its generality using rules extracted from existing traffic regulation documents.

5.2 Background and Motivation

This section surveys the state-of-the-art of safety policy development for autonomous driving and points out a representative safety policy standard, traffic laws, as our primary focus of safety verification.

5.2.1 AV Safety Policy

We survey some common safety polices recently proposed to guide the development of AV software systems. As one of the most well recognized safety guidance for AVs, Figure 5.1 illustrates the federal policy for highly automated vehicles or HAVs (i.e., SAE Levels 3-5 vehicles with automated systems that are responsible for monitoring the driving environment as defined by SAE J3016) issued by the U.S. Department of Transportation (DOT). In particular, federal, state and local laws for transportation need to be complied with by all HAV systems on the vehicles. Also, operating design domain (ODD), object and event detection and response (OEDR) and minimum risk condition (MRC) should be explicitly defined for each HAV system, with their details left to individual HAV vendors

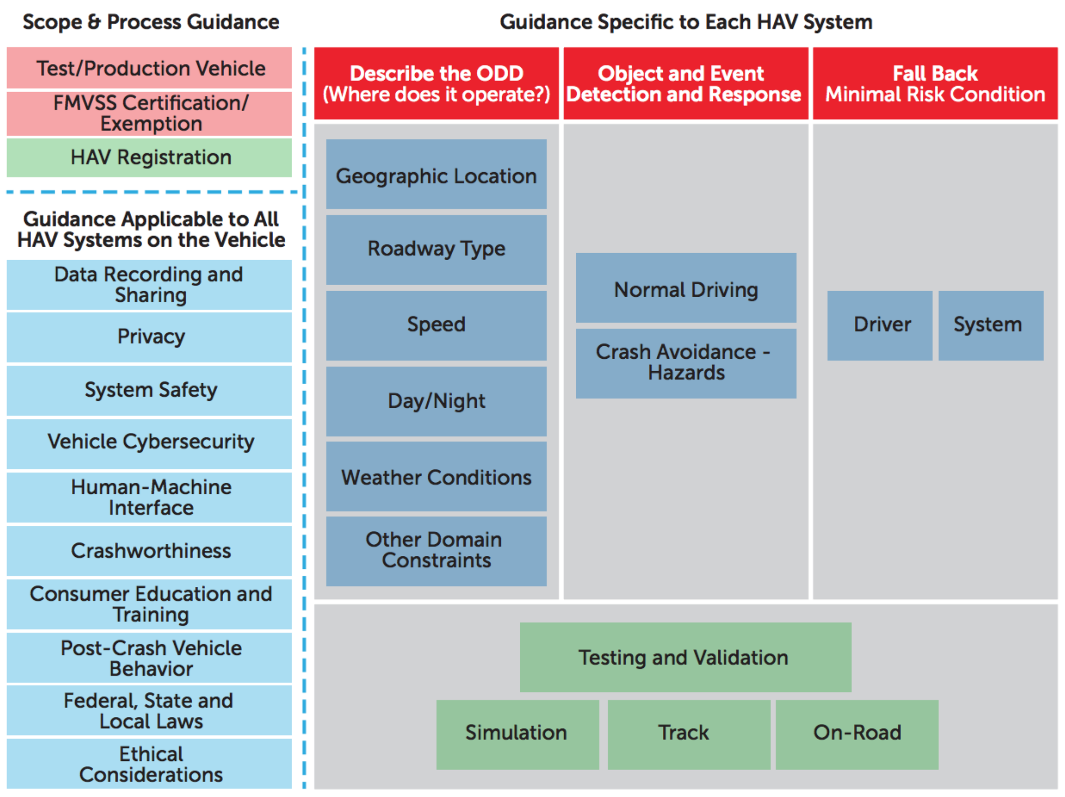

Scope & Process Guidance
Guidance Specific to Each HAV System
Test/Production Vehicle
FMVSS Certification/ Exemption
HAV Registration
Guidance Applicable to All HAV Systems on the Vehicle
Data Recording and Sharing
Privacy
System Safety
Vehicle Cybersecurity
Human-Machine Interface
Crashworthiness
Consumer Education and Training
Post-Crash Vehicle Behavior
Federal, State and Local Laws
Ethical Considerations
Describe the ODD (Where does it operate?)
Object and Event Detection and Response
Fall Back Minimal Risk Condition
Geographic Location
Roadway Type
Speed
Day/Night
Weather Conditions
Other Domain Constraints
Normal Driving
Crash Avoidance - Hazards
Driver
System
Testing and Validation
Simulation
Track
On-Road

Figure 5.1: U.S. DOT's safety policy framework for highly automated vehicles [17]

to specify.

More recently, Mobileye releases Responsibility-Sensitive Safety (RSS) [186], a rigorous mathematical model formalizing an interpretation of the *Duty of Care* law. The Duty of Care states that an individual should exercise "reasonable car" while performing acts that could harm others. RSS is constructed by formalizing the following 'common sense" rules of human driving:

- Do not hit someone from behind.
- Do not cut-in recklessly.
- Right-of-way is given, not taken.
- Be careful of areas with limited visibility
- If you can avoid an accident without causing another one, you must do it.

AVerifier aims to support the verification of various safety rules, include above two categories, by taking their specifications composed by code-level constructs as input and analyzing them at the code level of target AV software. It also aims to provide strong guarantees of zero false negatives in compliance verification: once compliance of a rule is verified, zero violation of that rule is guaranteed in the software logic. Furthermore, an AV-specific policy abstraction is proposed to support safety requirement specification for verifying user-defined safety policies. Towards achieving this safety verification goal, we start by verifying AV software's compliance with traffic regulations designed by governments for public transportation safety. As suggested by the Baidu Apollo developer, such regulations also apply to autonomous driving vehicles and an autonomous driving vehicle should follow traffic regulations at all times [139]. In particular, we extract dozens of common safety rules defined in state traffic laws [45] and verify them in the Baidu Apollo software.

5.2.2 Motivating Example: Baidu Apollo

We studied the software logic for traffic rule rule enforcement software logic in Baidu Apollo, the one and only one open-source AV software code base that supports common traffic rules to the best of our knowledge, to guide our design of the safety verification approach. Figure 5.2 illustrates the execution flow of the planning module of Baidu Apollo (v3.0) and highlights the components (in red) that enforce common traffic rules during path planning.

The traffic rule enforcement logic supports common traffic objects, e.g., traffic lights, stop sign, crosswalk, speed limit, etc. and common driving actions, e.g., stop, follow, sidepass, overtake, lane-change, nudge, etc. At run time of planning, the rule enforcement is performed for each planning cycle (100ms) through `apollo::planning::TrafficDecider::Execute` using the up-to-date traffic scene/frame. When this function is invoked, a predefined set of rule objects is ap-

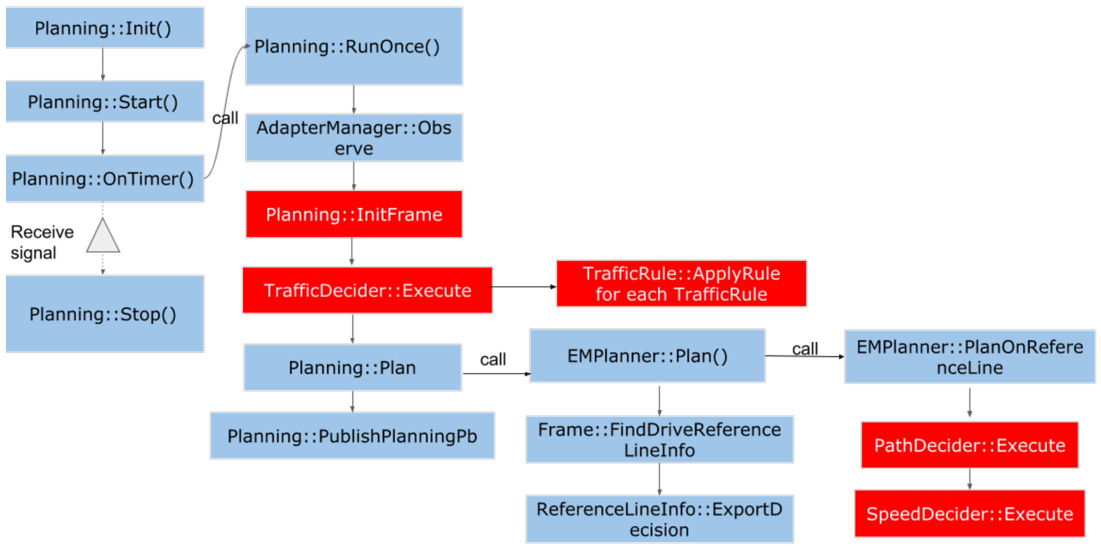


Figure 5.2: Overview of the traffic rule handling (highlighted in red) in Baidu Apollo (v3.0)

plied one by one through `apollo::planning::TrafficRule::ApplyRule` and per-object decisions are generated and combined as input costs and constraints for the optimization-based path computation by `apollo::planning::PathDecider` and `apollo::planning::SpeedDecider`.

5.3 AVerifier Design

This section presents the design overview and details of our proposed static program analysis and symbolic execution infrastructure to achieve the safety policy verification goal.

5.3.1 Overview

Figure 5.3 illustrates the key components and workflow of AVerifier. Given the source code of AV software and a list of safety rules as input, the code-level specification for each safety rule is generated based on a specification-to-code semantic mapping. The action in a rule specification is directly consumed by static analysis to identify the code-level predicates leading to the action, which are further compared with the predicates in the rule specification to detect potential violation. Furthermore, symbolic execution is applied to

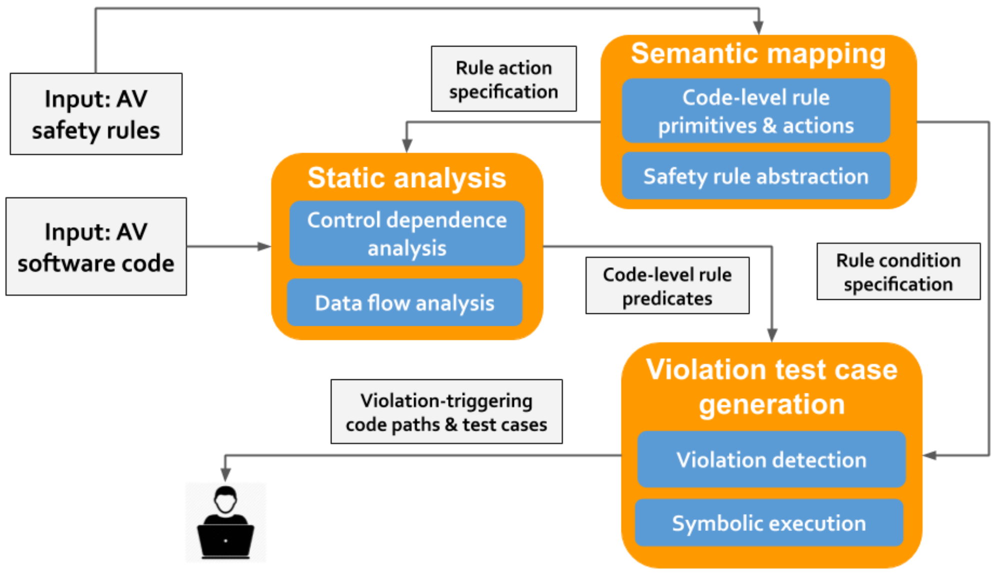

Input: AV safety rules
Rule action specification
Semantic mapping
Code-level rule primitives & actions
Safety rule abstraction
Input: AV software code
Static analysis
Control dependence analysis
Data flow analysis
Code-level rule predicates
Rule condition specification
Violation-triggering code paths & test cases
Violation test case generation
Violation detection
Symbolic execution

Figure 5.3: AVerifier framework overview

systematically explore the feasibility of violation cases.

5.3.2 Safety Rule Characterization

We collected and formulated 60 traffic rules from the handbook for getting driver licenses in Michigan [45] and characterize them based on the condition-action pattern. Table 5.1 summarizes the four main categories with rule patterns. According to our characterized rule patterns, safety driving rules defined in such traffic regulations commonly follow the "if-then" pattern (defined as *trigger-action* form). Certain traffic scenarios defined in triggering condition . To verify the consistency of a driving safety policy between its specification and code-level implementation, the first step is to identify the code-level predicates (a.k.a., "if" part) for the implemented policies in relevant modules of AV software, assuming that a set of driving decision action APIs used in the software is given. In terms of Baidu Apollo, traffic rule related policies are implemented in the planning module.

Rule pattern	**Example**
Don't apply *action()*	Do not use the bicycle lane as a right-turn lane
If *cond*, apply *action1()*, *action2()*, ...	If your car is disabled, pull far off the roadway, activate the emergency lights
Apply *action()* only if *cond1* and *cond2* and ...	When passing on the left, do not pass more than one vehicle at a time
Don't apply *action()* if *cond1* and *cond2* and ...	Do not pass if you are within 100 feet of an intersection

Table 5.1: Traffic rule categorization

5.3.3 Control Dependence Analysis

In program analysis, a statement S_2 has a *control dependence* on a preceding statement S_1 if the outcome of S_1 determines whether S_2 should be executed or not. The control dependence is defined by a control flow edge coming out of S_1 and reachable to S_2. A typical example of control dependence is the *if-else* branches: control dependences exist between the condition part of an *if* statement and the statements in its true/false branching bodies. Based on our safety driving rule characterization, the common "if-then" trigger-action form in safety rules can be extracted from control dependence analysis, where the "trigger" part is equivalent to the control dependence of a target program statement (a.k.a., "action").

Control dependence graph (CDG). In a control flow graph (CFG), a node d *dominates* a node n if every path from an entry node to n must go through d. By definition, every node dominates itself. The *dominance frontier* of a node d is the set of all nodes n such that d dominates an immediate predecessor of n, but d does not strictly dominate n. Control dependences are essentially the dominance frontier in the reverse graph of a CFG. Analogous to the definition of dominance above, a node z post-dominates a node n if all paths to an exit node of a CFG starting at n must go through z. Therefore, a common way to construct a CDG is computing post-dominance frontier of the CFG, and then reversing it to obtain a control dependence graph.

Post-dominator analysis. Given a CFG = $< V, E, Entry, Exit >$, assume $Exit$ is reachable from all V, a node p *post-dominates* v, if all paths from v to $Exit$ include p. Figure 5.4 presents the pseudo-code for constructing the post-dominance frontier.

In our problem context, we start from a set of predefined sinks (i.e., invocation of

```
for each X in a bottom-up traversal of the dominator tree do:
  PostDominanceFrontier(X) ← ∅
  for each Y ∊ Predecessors(X) do:
    if immediatePostDominator(Y) ≠ X:
      then PostDominanceFrontier(X) ← PostDominanceFrontier(X) ∪ {Y}
  done
  for each Z ∊ Children(X) do:
    for each Y ∊ PostDominanceFrontier(Z) do:
      if immediatePostDominator(Y) ≠ X:
        then PostDominanceFrontier(X) ← PostDominanceFrontier(X) ∪ {Y}
    done
  done
```

Figure 5.4: Algorithm for post-dominance frontier generation

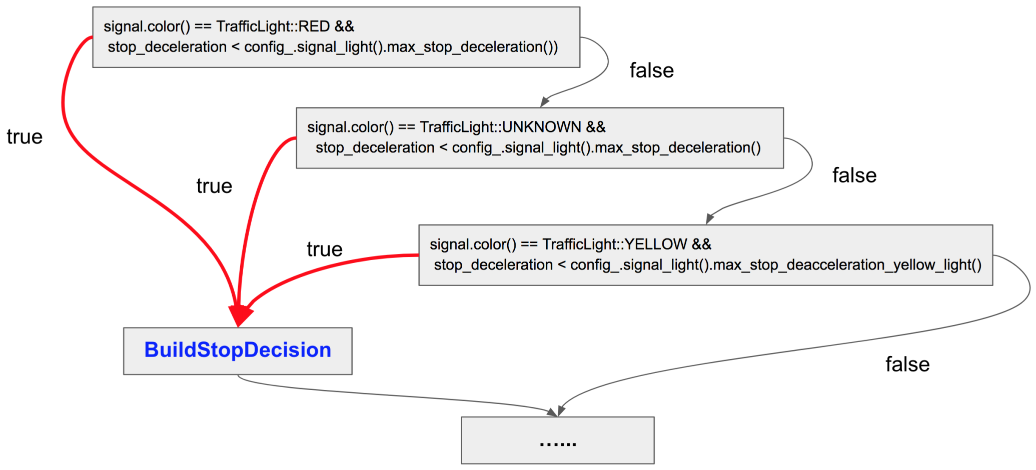
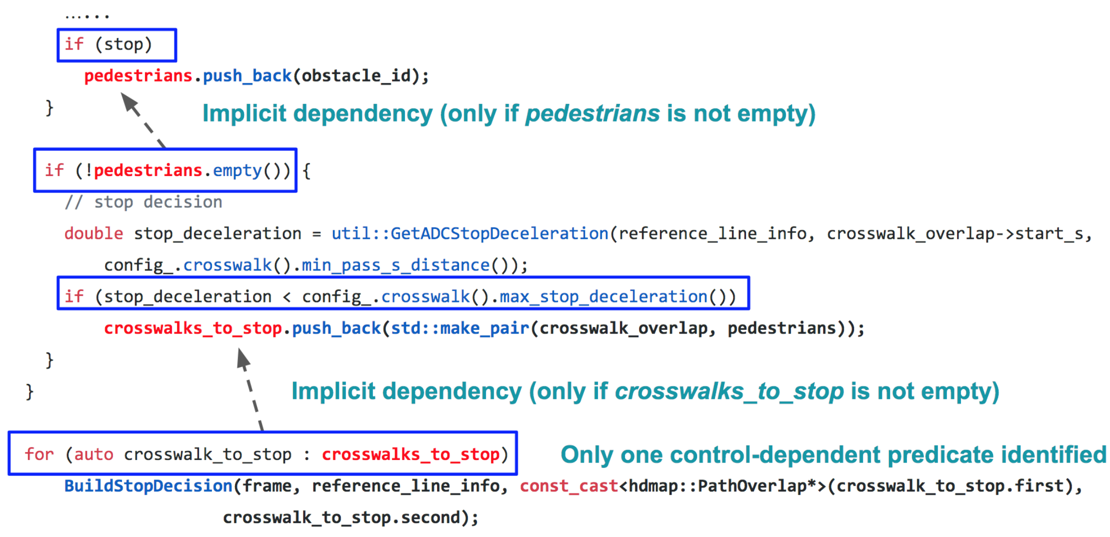

signal.color() == TrafficLight::RED &&
stop_deceleration < config_.signal_light().max_stop_deceleration())
false
true
signal.color() == TrafficLight::UNKNOWN &&
stop_deceleration < config_.signal_light().max_stop_deceleration()
true
false
true
signal.color() == TrafficLight::YELLOW &&
stop_deceleration < config_.signal_light().max_stop_deacceleration_yellow_light()
BuildStopDecision
false
......

Figure 5.5: Rule predicate extraction based on control dependence analysis (traffic light rule in Apollo)

control decision functions in Apollo) and trace its control dependences within a procedure. Figure 5.5 illustrates the control dependence for the stop decision function `apollo::planning::SignalLight::BuildStopDecision` in the traffic light rule handling in Apollo. Then the caller function is treated as an intermediate sink and the control dependences to all its invocations across the program are computed. This process is iterated until the program entries are hit. Finally, the set of rule predicates for a target sink is comprised of the union of control dependences identified from above analysis.



```
......
if (stop)
  pedestrians.push_back(obstacle_id);
}
if (!pedestrians.empty()) {
  // stop decision
  double stop_deceleration = util::GetADCStopDeceleration(reference_line_info, crosswalk_overlap->start_s,
      config_.crosswalk().min_pass_s_distance());
  if (stop_deceleration < config_.crosswalk().max_stop_deceleration())
      crosswalks_to_stop.push_back(std::make_pair(crosswalk_overlap, pedestrians));
}
}
for (auto crosswalk_to_stop : crosswalks_to_stop)
  BuildStopDecision(frame, reference_line_info, const_cast<hdmap::PathOverlap*>(crosswalk_to_stop.first),
                    crosswalk_to_stop.second);
```

Implicit dependency (only if *pedestrians* is not empty)
Implicit dependency (only if *crosswalks_to_stop* is not empty)
Only one control-dependent predicate identified

Figure 5.6: Implicit dependence not captured by control dependence analysis (crosswalk rule handling in Apollo)

5.3.4 Program Dependence Analysis

Control dependence in reality does not capture a complete set of rule predicates to manifest the triggering condition the runtime execution dependence in reality. Due to extensive use of composite data structures in the code of AV software, some code-level predicates corresponding to the high-level safety rule are connected through data dependence to an action sink (defined as *implicit dependence*). Figure 5.6 illustrates a motivating example based on the crosswalk rule handling logic in Baidu Apollo (v3.0).

To address this implicit dependence challenge, dataflow information, which can be computed based on the taint analysis, is embedded on top of the control dependence graph to form a program dependence graph, or PDG (illustrated in Figure 5.7). Similar to the previous rule predicate extraction on control dependence graph, starting from a predefined sink, we now trace its control and data dependence to identify intermediate sinks on a PDG and iterate above process on the intermediate sinks, until some program entries are hit. The final set of rule predicates consists of all the control-dependent predicates found from above analysis. According to our empirical study, the implicit dependence issues can be

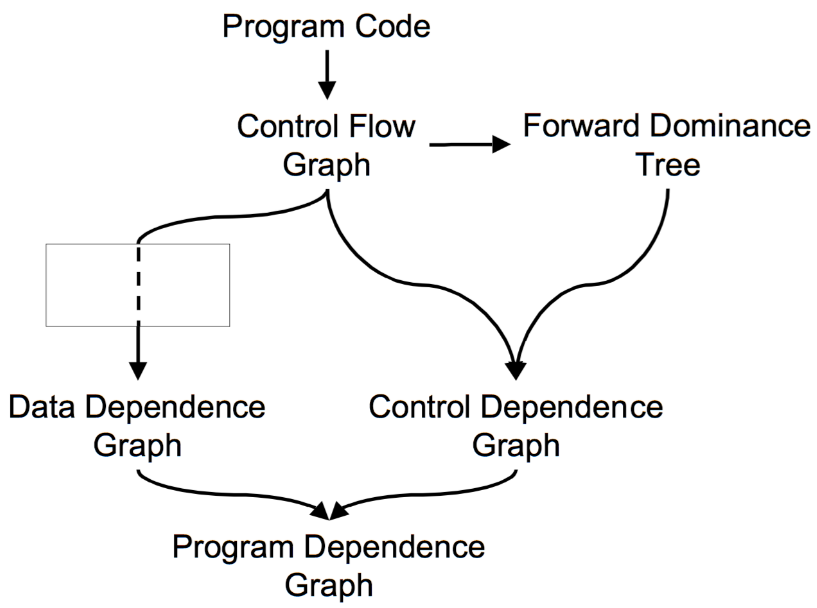


Figure 5.7: Overview of program dependence graph

fully addressed with PDGs so that a full set of rule predicates are extracted.

5.3.5 Detection of Rule Violation

Once the set of rule predicates determined from the PDG analysis, the constraints from each rule predicate are extracted and compared with conditions in the rule specification to detect potential violation. In our current prototype, this comparison is left to AV developers. If the conditions in the specified rule do not fall within the code-level predicate set, a potential violation of the input rule is flagged. For safety rules with “Do not X’ ’ action, we extract the code-level rule predicates leading to “X” action and take the complement constraint set of the rule predicates for comparison with the specified rules. According to our empirical evaluation and validation of traffic rules, the program dependence analysis is able to extract a complete set of code-level predicates. In the future, we plan to implement the symbolic execution support for AVerifier using KLEE [124] to enable systematic test case generations for violation-triggering paths and further validate the feasibility of detected violation. Our program dependence analysis narrows down to the code paths relevant to the safety rules of our interest and potentially mitigates the path explosion issue in symbolic execution.

5.3.6 Safety Rule Specification

To address the challenge in providing an expressive policy specification interface, we propose to define a generic abstraction for AV-specific policies in the trigger-action form, which consists of code-level policy primitives that are composable and extensible, and policy composition operators to enable flexible specification of code-level safety policies. First, AV-specific policy primitives are defined to capture a diverse set of traffic entities in common driving scenarios, e.g., static obstacles, moving objects (e.g., vehicle, pedestrian, bicyclist), and road signs (e.g., stop sign, traffic light, speed bump). These policy primitives will help construct the triggering condition of a policy. Second, driving control actions in a policy, e.g., stop, sidepass, overtake, are defined leveraging actuation APIs in the AV software code base. For example, the policy specification about stopping for a stop sign will have `stop sign` as a policy primitive in the triggering condition, and the `apollo::planning::StopSign::BuildStopDecision()` API as the action in Baidu Apollo [80]. We find that such specification is flexible enough to describe the 35 implemented rules out of the 60 traffic rules we extracted from the handbook for getting driver licenses in Michigan [45]. Meanwhile, it can also be conveniently mapped to code-level patterns in Baidu Apollo since the trigger-action form can be directly mapped to control dependencies, and the policy primitives and actuation APIs can be directly mapped to existing data structures in Baidu Apollo.

Specification-to-code traffic semantics. Table 5.2 lists of a set of AV-semantic data structures we identified from the Apollo code base. We assume that such code-level traffic semantics can be identified using some APIs in the code of AV software.

Policy specification with AV semantics. Table 5.3 shows some common policy primitives specified using above AV-semantic data structures to define the relative position between

Data structure	**Meaning**	**Attributes**
`PathObstacle.obstacle()`	An obstacle object	`Perception()`, `IsStatic()`, `IsVirtual()`
`PathObstacle.PerceptionSLBoundary()`	Physical boundary of an obstacle	`end_s()`, `start_s()`, `start_l()`, `end_l()`
`PathObstacle.st_boundary()`	Predicted speed of an obstacle over a period of time	`IsEmpty()`, `boundary_type()`, `min_s()`, `min_t()`, `max_s()`, `max_t()`
`PathObstacle.reference_line_st_boundary()`	Predicted speed of an obstacle over a period of time (on current reference line)	`IsEmpty()`, `boundary_type()`, `min_s()`, `min_t()`, `max_s()`, `max_t()`
`ReferenceLineInfo.reference_line()`	Planned driving path for AV	`Length()`, `map_path()`, `XYToSL()`, `HasOverlap()`
`ReferenceLineInfo.AdcSlBoundary()`	Physical boundary of AV	`end_s()`, `start_s()`, `start_l()`, `end_l()`
`ReferenceLineInfo.Lanes()`	Lanes used in planned driving path	`IsOnSegment()`
`VehicleStateProvider::instance()`	Av's current states	`linear_velocity()`, `x()`, `y()`, `z()`
`HDMapUtil::BaseMap()`	Road information in HD Map	`lane_table_`, `signal_table_`, ...

Table 5.2: Key data structures for traffic entities in Baidu Apollo (v3.0)

an AV and its surrounding objects. For example, a common traffic rule *stop behind a crosswalk for pedestrians on it* can be specified using *crosswalk*, *forthcoming* (i.e., in front of) and *pedestrian* primitives and the *stop* decision function as `If a` *`pedestrian`* `is on a` *`forthcoming crosswalk`*`, apply` *`stop`* `action`. Based on the code-level traffic primitives in Table 5.3, the code-level specification of this rule is presented in 5.4.

5.4 Evaluation

We perform the rule predicate extraction in the planning module of Baidu Apollo (v3.0) code base. We use the 35 implemented safety rules we extracted from the handbook for getting driver licenses in Michigan [45] to analyze the extracted predicates to find potential violation.

Traffic entity	Code-level specification
Object in front of AV	`PathObstacle.PerceptionSLBoundary().end_s() > ReferenceLineInfo.AdcSlBoundary().start_s()`
Object behind AV	`PathObstacle.PerceptionSLBoundary().start_s() < ReferenceLineInfo.AdcSlBoundary().end_s()`
Object on the left of AV	`PathObstacle.PerceptionSLBoundary().start_l() > ReferenceLineInfo.AdcSlBoundary().end_l()`
Object on the right of AV	`PathObstacle.PerceptionSLBoundary().end_l() < ReferenceLineInfo.AdcSlBoundary().start_l()`
Longitudinal distance ahead of AV	`PathObstacle.PerceptionSLBoundary().end_s() - ReferenceLineInfo.AdcSlBoundary().start_s()`
Lateral distance from the left of AV	`PathObstacle.PerceptionSLBoundary().start_l() - ReferenceLineInfo.AdcSlBoundary().end_l()`

Table 5.3: Code-level specification of common traffic entities

Policy primitive	Code-level trigger/action
Forthcoming crosswalk `c`	`c in reference_line().map_path().crosswalk_overlaps(), c.end_s() + config_.crosswalk().min_pass_s_distance() > ReferenceLineInfo.AdcSlBoundary().end_s()`
Pedestrian `p` on crosswalk `c`	`p.obstacle()->Perception().type() == PEDESTRIAN, p.obstacle()->Perception().x() > c.start_l(), p.obstacle()->Perception().x() < c.end_l(), p.obstacle()->Perception().y() > c.start_s(), p.obstacle()->Perception().y() < c.end_s()`
Stop decision	`AddLongitudinalDecision(p, object_decision=stop)`

Table 5.4: Code-level specification of rule: stop behind a crosswalk for pedestrians on it

5.4.1 Analysis Setup

Table 5.5 summarizes the decision functions as our analysis sinks for the safety rule verification in Apollo v3.0.

Decision function	Action type
`PathObstacle::AddLateralDecision`	sidepass, nudge
`PathObstacle::AddLongitudinalDecision`	stop, yield, follow, overtake, ignore
`ReferenceLineInfo::ExportTurnSignal`	Left/right turn signal
`PathDecision::MergeWithMainStop`	stop
`SpeedDecider::AppendIgnoreDecision`	ignore
`ReferenceLine::AddSpeedLimit`	speed limit
`ReferenceLineInfo::SetJunctionRightOfWay`	right-of-way

Table 5.5: Decision functions in Baidu Apollo (v3.0)

5.4.2 Rule Inconsistency Findings

We highlight some detected inconsistency findings between the specification and code-level implementation detected based on our current analysis. The current validation of the inconsistency requires human efforts. As future work, we plan to explore symbolic execution to automate the generation of violation-triggering test cases. We present the 2 cases that have been confirmed inconsistent with the corresponding safety rules as follows from our extensive code study and validation from different sources.

Rule: Slow down to 15mph when approaching a speed bump. We specify this safety rule as: for a *forthcoming speed bump*, if AV is approaching it within a distance of `min_slow_down_speedbump_distance` and has a speed over `max_speedbump_speed_limit`, invoke `ReferenceLine::AddSpeedLimit`. Using `ReferenceLine::AddSpeedLimit` as an analysis sink, we find that the code-level rule predicates leading to this sink does not contain the speed bump check. Its code-level specification in Table 5.6). Through manual validation, we confirm that this rule is not enforced in Baidu Apollo v3.0 code base. According to our further study, this missing condition check has been added into a later version (v3.5).

Policy primitive	**Code-level trigger/action**
Approaching a speed bump s	`s in reference_line().map_path().speed_bump_overlaps(), s.start_s() > ReferenceLineInfo.AdcSlBoundary().end_s(), s.start_s() - ReferenceLineInfo.AdcSlBoundary().end_s() < min_slow_down_speedbump_distance`
AV speed over limit	`VehicleStateProvider.linear_velocity() > max_speedbump_speed_limit`
Speed limt decision	`AddSpeedLimit(s.start_s(), s.end_s(), max_speedbump_speed_limit)`

Table 5.6: Code-level rule specification: slow down to 15mph when approaching a speed bump

Rule: Do not pass if you are within 100 feet of an intersection. Using `PathObstacle::AddLateralDecision` as an analysis sink, we find that the code-level rule predicates leading to this sink with `side pass` object decision contains the constraint that the AV is within 100 feet of an intersection. Through manual validation, we confirm that this rule is not enforced in Baidu Apollo v3.0 code base. According to our further study, this missing condition check has been added into a later version (v3.5).

5.5 Discussion and Future Work

In this section, we discuss the current limitation with our approach, and propose immediate and potential future work following this research direction.

5.5.1 Improvement on Policy Specification-to-Code Mapping

The major limitation of the AVerifier approach is the need of defining the traffic entities for composing policy primitives in safety rule specification and constructing their semantic mapping to the code-level implementation of AV software. This in reality may require AV developers' input to customize the mapping for target AV software and involve certain amounts of manual efforts for building the mapping. However, this is a one-time effort and can be potentially facilitated with the help of API documentations available with AV software. In the future, we plan to evaluate the actual amount of efforts required in more comprehensive ways and evaluate the usability and effectiveness tradeoff caused by this safety verification workflow.

5.5.2 Symbolic Execution Support

An engineering limitation of the current AVerifier prototype is the lack of support of symbolic execution for exploring the feasibility of paths relevant to a detected violation and generating concrete test cases to validate them in realistic traffic scenarios. We plan to implement the symbolic execution support for AVerifier using KLEE [124] to enable systematic test case generations for violation-triggering paths and further validate the feasibility of detected violation. Our program dependence analysis narrows down to the code paths relevant to the safety rules of our interest and can potentially reduce the chance of path explosion in symbolic execution.

5.5.3 Improvement on Violation Detection and Validation

Given the constraints in the rule predicates extracted from the program dependence analysis, the current approach requires AV developers to determine any mismatch between the extracted constraints and those in the rule specification. This manual checking process to find potential violation can be nontrivial and may affect the usability of AVerifier. To automate this, some road traffic and driving semantics (e.g., speed, position, etc.) need to be considered to help determining the semantic relationship (e.g., full inclusion, partial overlap, etc.), which can directly guide the violation detection outcome. We admit that this is another key research challenge and plan to explore how to define such semantic comparison in the future. Also, the detected inconsistency or violation of target safety rules needs to be validated in a more rigorous and automated way. We plan to leverage symbolic execution and simulation tools (e.g, Baidu Apollo's built-in SimControl) to generate and reproduce the violation-triggering cases in realistic traffic scenarios.

5.6 Summary

In this work, we develop AVerifier, a static program analysis framework towards verifying the safety requirement compliance in AV software with completeness guarantees and zero false negatives in compliance verification. Specifically, We design and implement a program dependence analysis framework to help AV developers perform static detection of safety requirement noncompliance problems in AV software. Our analysis uncovers the noncompliance with some standardized safety requirements in an early version of Baidu Apollo. To allow expressive specification of safety requirements, a specification interface is proposed based on a unique abstraction of the rich road traffic and driving semantics and a semantic mapping that relates each semantic entity to its code-level implementation in AV software. We validate that this interface enables AV developers to express a broad range of common safety rules from existing traffic regulation documents in a composable and flexible manner.

CHAPTER VI

Related Work

6.1 Performance Testing and Diagnosis for Smart Systems

Mobile app testing: Functionality bugs, crashes or performance issues in mobile apps can be uncovered through automated UI testing [110, 111, 169, 116, 182, 152, 126, 150, 170, 172, 165], runtime analysis [161, 183, 206, 126, 157, 195, 123] or static analysis [163, 168]. Static analysis is not proper for uncovering runtime cause of performance issues since it does not capture runtime contexts. PerfProbe is complementary to existing app testing and profiling tools. PerfProbe can be used to monitor user interactions with performance issues detected from testing tools and perform further trace-based diagnosis to help understand the root cause of the issues.

Mobile performance monitoring: PerfProbe complements existing profiling-based diagnosis systems in providing an automated systematic approach that performs lightweight profiling and holistic analysis of app and OS-layer runtime events to guide the root cause diagnosis of a broad category of real-world performance issues, which is less extensively explored. ProfileDroid [198] profiles cross-layer information including static app specification, user interaction, operating system, and network of individual Android apps to detect performance problems in Android platform, but it lacks a systematic approach to diagnose the cause of performance problems. ARO [180] exposes the cross-layer interaction in the network stack to understand energy and radio resource usage of mobile apps, but does not focus on diagnosing user-perceived latency problems. It can be potentially integrated into PerfProbe for understanding the impact of radio resource usage on the performance. AppInsight [183] instruments app binary for performane monitoring and profiling in the wild. PerfGuard [160] proposes an automated, lightweight approach to instrument binary code for performance monitoring and diagnosis in production environments. Panappticon [206] instruments Android OS and framework to track OS-wide events but cannot provide easy-to-reason code-level hints to app developers. Also, PerfProbe's cross-layer characterization can provide systematic root cause reasoning and validation on potentially runtime-expensive operations detected by existing tools [168, 205, 195]. PerfProbe leverages Panappticon's instrumentation framework [206] to record OS events and Traceview [72, 51] to capture an app's call stack, but provides a new diagnosis approach by systematically associating such cross-layer runtime information to gain holistic understanding on the cause of performance slowdown. App instrumentation [183, 149, 160] may achieve lower monitoring overhead compared to Traceview, but requires specification from app developers and is thus unscalable for systematic localization of code-level performance variance (i.e., critical function) in arbitrary apps.

Cross-layer performance analysis: Cross-layer analysis was applied to investigate the performance of wearable systems [167] and smartphone platforms [180, 198, 126]. ARO [180] mainly focuses on radio resource efficiency problems rather than app QoE, while QoE Doctor cannot break down app latencies into more fine-grained operations, such as disk access and inter-process communication that PerfProbe can address, due to the lack of system or application event profiling.

Performance diagnosis using machine learning: Previous works [132, 173] apply machine learning techniques on system logs to diagnose performance problems in distributed systems. Decision tree is also used for per-application QoS-to-QoE mapping for QoE inference of mobile apps [175]. Though PerfProbe also leverages decision tree learning, the use of decision tree in the two-step analysis is new and different from them. Gist [158] relies on hardware features for extracting control flow traces to sketch failure predicting events that have the highest positive correlation with the occurrence of failures. However, these hardware features that Gist relies on are not widely available in mobile processors and is thus not applicable to diagnosing mobile app performance problems.

6.2 Security Testing and Defense for Smart Systems

Permission models & overprivilege detection: Previous works have discovered various overprivilege problems in smartphone [142, 141] and smart home systems [143] through static and dynamic analysis. Other work characterizes and mitigates context-based overprivilege issues with smart home systems a [155]. Our overprivilege problem is different from them in two aspects. First, previous overprivilege problems happen in systems with regular user interactions while human interaction may not exist in AV systems. Second, different from previous overprivilege in API access, our overprivilege problem is in the access to message fields during the publish-subscribe communication of AV systems. Due to these differences, we develop a system-level and module-transparent solution to perform fine-grained permission control on this communication channel.

Static analysis for vulnerability detection: Static analysis is a common approach for vulnerability detection in various software systems, but usually requires specific accommodation to the problem domain. To address complexity due to the event and component-based nature in smartphone systems, inter-component communication, life-cycle awareness and inter-component data flow graph [176, 114, 197, 154] are built on top of standard dataflow analysis to uncover common vulnerabilities in smartphone systems. Furthermore, call-graph analysis with pruning heuristics is used to discover inconsistency in security policy enforcement in Android framework [187]. Our work contributes to this area in presenting new insights on how standard dataflow analysis techniques can be adapted to the virtual functions and asynchronous programming models for security analysis in the AV software system domain.

Security policy enforcement: There are two major approaches for security policy enforcement. One approach is providing developers with access control APIs [184]. Another is automatic policy generation and enforcement. Previous work performs automated authorization hook placement on Linux security modules framework to guard kernel functions [145]. AutoISES [192] automatically infers security policies by statically analyzing source code and uses them to detect security violations. Aurasium [203] automatically repackages arbitrary Android apps to attach user-level sandboxing and policy enforcement code. Different from previous work, AVGuardian targets at the publish-subscribe message channel rather than code level to enforce fine-grained access control policies generated from static overprivilege detection under the module-compromise threat model, where code-level policy placement and enforcement can be bypassed.

Vehicle attack surface analysis: Attacks surface analysis has been conducted on in-vehicle network of modern automobiles [162, 125, 130], vehicular applications [137], perception sensors for autonomous driving systems [179], and connected vehicular communication channel [127]. Our work contributes to this area in discovering overprivilege problems with the publish-subscribe communication channel as new attack surfaces in autonomous vehicle systems and proposing a systematic approach to detect these attack surfaces. One major defense solution to existing in-vehicle attacks is intrusion detection. Fingerprinting-based solutions based on time profiles of periodic messages sent from Electronic Control Units (ECUs) [129] or voltage profiles of ECUs [131], are shown to be effective in identifying an attacker-controlled ECU. Our work proposes a different mitigation solution to the overprivilege problem through policy enforcement and provides new insight for enabling fine-grained permission control on AV software.

6.3 Requirement Verification Techniques

Static analysis approach: Meta-level compilation (MC) is proposed to write system-specific compiler extensions that automatically check the source code for violations of system rules [138]. Programmers can write system-specific compiler extensions to enable system-specific static analysis to find security errors that violate certain system rules [115]. Rules for driver API usage can be verified and temporal safety properties of sequential C programs I can checked thoroughly [118]. Furthermore, static analysis can be used to identify a small portion of paths relevant to rules and apply symbolic execution them for verifying system rules with better scalability [135]. Our safety requirement focuses on verifying different safety rules for autonomous driving and requires AV-specific traffic modeling to bridge the semantic gap between specification and code implementation.

Model checking approach: Model checking is a technique for automatically verifying correctness properties of finite-state systems. In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language. Model checking software based on the abstract-check-refine paradigm [151] or predicate abstraction techniques [119] have been shown effective in verifying desired properties and find violation in system software. Furthermore, formal techniques are applied to verify the design requirements of real-world safety-critical systems [177]. However, existing approaches cannot be directly applied to the verification of safety requirements for autonomous driving due to the lack of system-specific semantic mapping in requirement specification. Our safety verification work on AV software takes the first step to define AV-specific semantics for specifying system-specific system rules.

CHAPTER VII

Conclusion and Future Work

In this chapter, we conclude by highlighting the contributions of this dissertation and discuss future research directions based on the current research contribution in this dissertation.

7.1 Concluding Remarks

As various forms of smart end systems are emerging and making significant influence to our daily work and life, it becomes critical to assure expected performance and robustness with their software as they operate in the real world. However, in reality, the required performance, security and safety guarantees have never been thoroughly tested and verified on the well developed smartphone systems, not to mention the emerging autonomous vehicle platforms.

This dissertation proposes practical and systematic software analysis solutions towards testing and verifying the performance, security and safety requirements of the software for smart end systems. Specifically, we demonstrate that automated program analyses based on 1) static program analysis for achieving completeness guarantees of analyzing program behaviors and 2) runtime program profiling for capturing runtime conditions of program execution, can achieve systematic requirements testing and verification for smart end systems and significantly reduce manual efforts. This dissertation contributes to the requirements testing and verification of smart end systems in following aspects: (1) effectively test performance requirements and diagnose the cause of performance slowdown through lightweight monitoring of and systematic performance characterization based on cross-layer runtime events, (2) systematically detect noncompliance with important security principles (e.g., publish-subscribe overprivilege vulnerability) through systematic program analysis and mitigate security vulnerabilities through policy enforcement, and (3) systematically verify the compliance with safety requirements on the mission-critical components (e.g., AV's driving decision control) of smart end systems.

7.2 Future Work

AVerifier requires defining the traffic entities for composing policy primitives in safety rule specification and constructing their semantic mapping to the code-level implementation of AV software. This in reality may require AV developers' input to customize the mapping for target AV software and involve certain amounts of manual efforts for building the mapping. An interesting question is to evaluate the actual amount of efforts required in more comprehensive ways and its resulted usability and effectiveness tradeoff. Another future direction is to involve road traffic and driving semantics (e.g., speed, position, etc.) for building the semantic relationship (e.g., full inclusion, partial overlap, etc.), which can directly guide the violation detection outcome. In the future, we plan to implement the symbolic execution support for AVerifier using KLEE [124] to enable systematic test case generations for violation-triggering paths and further validate the feasibility of detected violation, e.g., using simulation in realistic traffic scenes.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] AI Camera Demo App. https://caffe2.ai/docs/AI-Camera-demo-android.html.

[2] Android Open Source Project. https://source.android.com.

[3] Android OS Architecture. http://www.techplayon.com/android-os-architecture.

[4] Apollo Github commit log – canbus: fix MonitorBuffer crash problem when AdapterManager is not initialized. https://github.com/ApolloAuto/apollo/commit/751e9ab63da7a2240dc54cb2c7c7cbde1b9a4ede.

[5] Apollo Github commit log – Conti radar driver : fix security issue. https://github.com/ApolloAuto/apollo/commit/5bca95d8d69b5a82269b4725f03a9010525b0b3f.

[6] Apollo Github commit log – Driver: fixed a bug in canbus socket_can_client_raw.cc. https://github.com/ApolloAuto/apollo/commit/c8abb2f80d7c41bfc5f542e3a55ebd2181c79e8a.

[7] Apollo Github commit log – drivers: gnss fix array index out of bounds. https://github.com/ApolloAuto/apollo/commit/39eacd9ced6e69ab6f87336e171962a98c3c5c85.

[8] Apollo Github commit log – drivers: velodyne fix index bounds. https://github.com/ApolloAuto/apollo/commit/5c13d1f5ce8c3492f912e1b76ec9538c2d64c938.

[9] Apollo Github commit log – Fix driver safety scan bug. https://github.com/ApolloAuto/apollo/commit/8bd16f986c38619fb5f4a9c6b330139dd705c96a.

[10] Apollo Github commit log – Fix driver safety scan bug. https://github.com/ApolloAuto/apollo/commit/3ca9d93a9a41e42b7d5867a96271b985b7caf603.

[11] Apollo Github commit log – fix sercurity issues in driver. https://github.com/ApolloAuto/apollo/commit/b80379957a72c39b1412e18b6bce1a6ccbb16eec.

[12] Apollo Github commit log – Fixed a couple security bugs. https://github.com/ApolloAuto/apollo/commit/d8da92557d00550f2ca17bf9bd583f385dbbc67b.

[13] Apollo Github commit log – GNSS: fixed some null pointer issue. https://github.com/ApolloAuto/apollo/commit/f9db5ebf699fd82261f1911b519ae16ab6874ee0.

[14] ApolloAuto: An open autonomous driving platform. https://github.com/ApolloAuto/apollo.

[15] Apolong: The 1st L4 Autonomous Driving Mini Bus Achieved Mass Production. http://apollo.auto/cooperation/detail_en_07.html.

[16] App Annie - Top Apps on Google Play. https://www.appannie.com/apps/google-play/top/.

[17] Automated Driving Systems 2.0: A Vision for Safety. https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf.

[18] Autoware: Open-Source To Self-Driving. https://github.com/CPFL/Autoware.

[19] Autoware's AS package. https://github.com/CPFL/Autoware/blob/1.9.1/ros/src/actuation/vehicles/packages/as/nodes/pacmod_interface/pacmod_interface.cpp#L87.

[20] Autoware's autoware_connector package. https://github.com/CPFL/Autoware/blob/1.9.1/ros/src/computing/perception/localization/packages/autoware_connector/nodes/can_odometry/can_odometry_core.cpp#L125.

[21] Autoware's obj_reproj package. https://github.com/CPFL/Autoware/blob/1.9.1/ros/src/computing/perception/detection/lidar_tracker/packages/obj_reproj/nodes/obj_reproj/obj_reproj.cpp#L318.

[22] Autoware's waypoint_follower package. https://github.com/CPFL/Autoware/blob/1.9.1/ros/src/computing/planning/motion/packages/waypoint_follower/nodes/twist_gate/twist_gate.cpp#L215.

[23] Baidus autonomous driving technology finds new application in urban cleaning. https://technode.com/2018/09/28/baidu-autonomous-driving.

[24] c:geo. https://play.google.com/store/apps/details?id=cgeo.geocaching.

[25] Chassis message definition. https://github.com/ApolloAuto/apollo/blob/v3.0.0/modules/canbus/proto/chassis.proto.

[26] CIDI equipped with Apollo 2.5 in running high-speed trucks. http://apollo.auto/cooperation/detail_en_06.html.

[27] CNET. https://play.google.com/store/apps/details?id=com.treemolabs.apps.cnet.

[28] Collections of Apollo Platform Software. https://github.com/ApolloAuto/apollo-platform.

[29] CompensatorNodelet. https://github.com/ApolloAuto/apollo/tree/v2.5.0/modules/drivers/velodyne/velodyne_pointcloud/src.

[30] Dont Let Identity Thieves Take You for a Ride. https://www.experian.com/blogs/ask-experian/dont-let-identity-thieves-take-you-for-a-ride.

[31] Download Manager. https://play.google.com/store/apps/details?id=com.acr.androiddownloadmanager.

[32] Event-Based Tracing to Measure Android Application and Platform Performance. https://github.com/EmbeddedAtUM/panappticon.

[33] Faster loading for all guides - just show the first page of results. https://github.com/inaturalist/iNaturalistAndroid/commit/ea6d2892d545cc6ae203edcf6823f0200d446fd0.

[34] First Impressions Count: Boost Your App's Start-Up Time. https://developer.amazon.com/blogs/post/Tx1RLL07TPIH1RJ/First-Impressions-Count-Boost-Your-App-s-Start-Up-Time.html.

[35] Flipp - Weekly Ads & Coupons. https://play.google.com/store/apps/details?id=com.wishabi.flipp.

[36] Free-Fall: Hacking Tesla from Wireless to CAN Bus. https://www.blackhat.com/docs/us-17/thursday/us-17-Nie-Free-Fall-Hacking-Tesla-From-Wireless-To-CAN-Bus-wp.pdf.

[37] fromPCLPointCloud2 definition. http://docs.pointclouds.org/1.7.0/conversions_8h_source.html#l00167.

[38] Geohash Droid. https://play.google.com/store/apps/details?id=net.exclaimindustries.geohashdroid.

[39] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[40] Google Play. https://play.google.com/store.

[41] Google play: number of android app downloads 2010-2016. https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

[42] Google Translate. https://play.google.com/store/apps/details?id=com.google.android.apps.translate.

[43] Guide on Nuvo-6108GC Installation. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/IPC/Nuvo-6108GC_Installation_Guide.md.

[44] Hackers Remotely Kill a Jeep on the Highway With Me in It. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway.

[45] Handbook for Getting Driver Licenses in Michigan: What Every Driver Must Know. https://driving-tests.org/wp-content/uploads/2018/05/MI_wedmk_16312_7.pdf.

[46] H&M. https://play.google.com/store/apps/details?id=com.hm.

[47] iNaturalist. https://play.google.com/store/apps/details?id=org.inaturalist.android.

[48] Increase Read Cache for better SD Card access. https://forum.xda-developers.com/showthread.php?t=1010807.

[49] Increase the read/write speed of the SD card on your rooted Android tablet. https://goo.gl/PNyYHa.

[50] Increase Your SD Card Read Speeds By 100-200% With A Simple Tweak. https://goo.gl/Hpce69.

[51] Inspect CPU activity with CPU Profiler. https://developer.android.com/studio/profile/cpu-profiler.

[52] Instruments User Guide. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/.

[53] Intermittent slowdown for loading nearby cache. https://github.com/cgeo/cgeo/issues/6632.

[54] Intermittently long waiting time for search result to be displayed. https://github.com/tomahawk-player/tomahawk-android/issues/86.

[55] Intermittently slowness for mailbox refreshing. https://github.com/k9mail/k-9/issues/2575.

[56] JSON Sudoku Solver. https://play.google.com/store/apps/details?id=com.musevisions.android.SudokuSolver.

[57] K-9 Mail. https://play.google.com/store/apps/details?id=com.fsck.k9.

[58] Libgcrypt. https://gnupg.org/related_software/libgcrypt.

[59] Linux CPUFreq User Guide. https://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt.

[60] Loading Large Bitmaps Efficiently. https://developer.android.com/topic/performance/graphics/load-bitmap.html.

[61] Meitu - Beauty Cam, Easy Photo Editor. https://play.google.com/store/apps/details?id=com.mt.mtxx.mtxx.

[62] Momenta: Apollo 1.5 Set Line Autopilot Solution. http://apollo.auto/cooperation/detail_en_03.html.

[63] New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars. https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

[64] NimbleDroid Blog. http://blog.nimbledroid.com/.

[65] Number of android applications. http://www.appbrain.com/stats/number-of-android-apps.

[66] OfferUp - Buy. Sell. OfferUp. https://play.google.com/store/apps/details?id=com.offerup.

[67] OProfile. http://oprofile.sourceforge.net/.

[68] Over-the-Air: How we Remotely Compromised the Gateway, BCM, and Autopilot ECUs of Tesla Cars. https://i.blackhat.com/us-18/Thu-August-9/us-18-Liu-Over-The-Air-How-We-Remotely-Compromised-The-Gateway-Bcm-And-Autopilot-Ecus-Of-Tesla-Cars-wp.pdf.

[69] Pandora: All-in-One Sensing Kit. http://www.hesaitech.com/en/pandora.html.

[70] PerfProbe Case Study. https://sites.google.com/site/perfprobe/case.

[71] PhoneLab: A Smartphone Platform Testbed. https://www.phone-lab.org/.

[72] Profiling with Traceview and dmtracedump. https://developer.android.com/studio/profile/traceview.html.

[73] Protocol Buffers - Google's data interchange format. https://github.com/google/protobuf.

[74] Riot.im - open team collaboration. https://play.google.com/store/apps/details?id=im.vector.alpha.

[75] ROS (Robot Operating System). http://wiki.ros.org.

[76] ROS Technical Overview. http://wiki.ros.org/ROS/Technical%20Overview.

[77] ROS2 Security. https://discourse.ros.org/t/ros2-security/2273.

[78] Self-Driving Ubers. https://www.uber.com/cities/pittsburgh/self-driving-ubers.

[79] sensor_msgs/PointCloud2 Message. http://docs.ros.org/melodic/api/sensor_msgs/html/msg/PointCloud2.html.

[80] SignalLight::BuildStopDecision() Function in Baidu Apollo. https://github.com/ApolloAuto/apollo/blob/master/modules/planning/traffic_rules/signal_light.cc#L230.

[81] Sina News. https://play.google.com/store/apps/details?id=com.sina.news.

[82] sklearn.tree.decisiontreeclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[83] Slow loading of ALL Guides tab. https://github.com/inaturalist/iNaturalistAndroid/issues/375.

[84] SROS. http://wiki.ros.org/SROS.

[85] SROS2: ROS2 on top of DDS-Security. https://github.com/ros2/sros2.

[86] SSE - Universal Encryption App. https://play.google.com/store/apps/details?id=com.paranoiaworks.unicus.android.sse.

[87] Suggestion for improving directory loading performance. https://github.com/vector-im/riot-android/issues/1473.

[88] Suggestion for improving map rendering performance during app launch. https://github.com/CaptainSpam/geohashdroid/issues/67.

[89] Suning Logistics and Baidu Apollo Partnering on Self-Driving Technology. https://www.auvsi.org/industry-news/suning-logistics-and-baidu-apollo-partnering-self-driving-technology.

[90] Tempering the Utopian Vision of the Mobility Revolution. https://www2.deloitte.com/insights/us/en/industry/automotive/advanced-vehicle-technologies-mobility-revolution.html.

[91] Tencent Keen Security Lab: Experimental Security Research of Tesla Autopilot. https://keenlab.tencent.com/en/2019/03/29/Tencent-Keen-Security-Lab-Experimental-Security-Research-of-Tesla-Autopilot.

[92] Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr/tesseract.

[93] Text Fairy (OCR Text Scanner). https://play.google.com/store/apps/details?id=com.renard.ocr.

[94] TF Detect. https://play.google.com/store/apps/details?id=org.tensorflow.app&hl=en_US.

[95] TFBroadcasterNodelet. https://github.com/ApolloAuto/apollo/tree/v2.5.0/modules/drivers/gnss/src/tf.

[96] tf/tfMessage Message. http://docs.ros.org/melodic/api/tf/html/msg/tfMessage.html.

[97] The OpenCAV Platform. http://www.andrew.cmu.edu/user/dingzhao/OpenCAV.html.

[98] The RAIL Performance Model. https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail.

[99] Tomahawk Player Beta. https://play.google.com/store/apps/details?id=org.tomahawk.tomahawk_android.

[100] uiautomator. http://android.xsoftlab.net/tools/help/uiautomator/index.html.

[101] Velodyne LiDAR. http://www.velodynelidar.com.

[102] Vine Camera. https://play.google.com/store/apps/details?id=co.vine.android.

[103] VLC for Android. https://play.google.com/store/apps/details?id=org.videolan.vlc.

[104] Volkswagen taps Baidu's Apollo platform to develop self-driving cars in China. https://www.reuters.com/article/us-volkswagen-autonomous/vw-taps-baidus-apollo-platform-to-develop-self-driving-cars-in-china-idUSKCN1N71J1.

[105] Volvo and Baidu join forces to mass produce self-driving electric cars in China. https://www.cnbc.com/2018/11/01/volvo-baidu-to-mass-produce-self-driving-electric-cars-in-china.html.

[106] Waymo. https://waymo.com.

[107] Where Am I? https://play.google.com/store/apps/details?id=com.ejelta.whereami.

[108] Your vehicle's VIN could reveal more than you want. https://www.wavy.com/10-on-your-side/investigative/special-report-privacy-breakdown/109991548.

[109] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-wesley Reading, 2007.

[110] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, 2012.

[111] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[112] B. Apollo. Sensor Data Sample. http://data.apollo.auto/help?name=data-sensor%20demostration-user%20guide&data_key=sensor&data_type=1&locale=en-us&lang=en, 2019.

[113] 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 *PLDI*, 2014.

[114] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[115] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, 2002.

[116] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, 2013.

[117] J. Bacon, D. M. Eyers, J. Singh, and P. R. Pietzuch. Access control in publish/-subscribe systems. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, 2008.

[118] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers.

In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, 2006.

[119] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, 2002.

[120] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based Access Control for Publish/Subscribe Middleware Architectures. In *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*, DEBS '03, 2003.

[121] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, and U. Muller. Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. *CoRR*, 2017.

[122] B. Breiling, B. Dieber, and P. Schartner. Secure communication for the robot operating system. In *2017 Annual IEEE International Systems Conference*, SysCon 2017, 2017.

[123] M. Brocanelli and X. Wang. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *Proc. of the 2014 ACM European Conference on Computer Systems*, EuroSys '18, 2018.

[124] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, 2008.

[125] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, 2011.

[126] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis. In *Proc. of IMC*, 2014.

[127] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. L. Liu. Exposing Congestion Attack on Emerging Connected Vehicle based Traffic Signal Control. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS '18, 2018.

[128] B. Chess and G. McGraw. Static Analysis for Security. In *IEEE Security & Privacy*, 2004.

[129] K.-T. Cho and K. Shin. Fingerprinting Electronic Control Units for Vehicle Intrusion Detection. In *Proceedings of the 25th USENIX Security Symposium*, SEC'16, 2016.

[130] K.-T. Cho and K. G. Shin. Error Handling of In-vehicle Networks Makes Them Vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, 2016.

[131] K.-T. Cho and K.-G. Shin. Viden: Attacker Identification on In-Vehicle Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, 2017.

[132] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, 2004.

[133] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security*, 1998.

[134] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, 2010.

[135] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying Systems Rules Using Rule-directed Symbolic Execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[136] B. Dieber, S. Kacianka, S. Rass, and P. Schartner. Application-level security for ROS-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[137] D. Dominic, S. Chhawri, R. M. Eustice, D. Ma, and A. Weimerskirch. Risk Assessment for Cooperative Automated Driving. In *Proceedings of the 2Nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, CPS-SPC '16, 2016.

[138] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-specific, Programmer-written Compiler Extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*, OSDI'00, 2000.

[139] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong. Baidu Apollo EM Motion Planner. *CoRR*, 2018.

[140] T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letter*, 2006.

[141] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[142] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps'11, 2011.

[143] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, SP'16, 2016.

[144] I. Foster, A. Prudhomme, K. Koscher, and S. Savage. Fast and Vulnerable: A Story of Telematic Failures. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, 2015.

[145] V. Ganapathy, T. Jaeger, and S. Jha. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, 2005.

[146] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[147] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, 2015.

[148] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[149] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, 2013.

[150] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[151] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, 2002.

[152] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proc. of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, 2014.

[153] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[154] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao. Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, EuroSP '17, 2017.

[155] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platform. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS '17, 2017.

[156] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, 2005.

[157] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu. DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[158] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[159] N. Khadke, M. P. Kasick, S. P. Kavulya, J. Tan, and P. Narasimhan. Transparent system call based performance debugging for cloud computing. In *Proceedings of the 2012 Workshop on Managing Systems Automatically and Dynamically*, 2012.

[160] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu. PerfGuard: Binary-centric Application Performance Monitoring in Production Environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[161] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu. IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, 2014.

[162] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, 2010.

[163] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic Performance Prediction for Smartphone Applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, 2013.

[164] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010.

[165] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[166] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005.

[167] R. Liu and F. X. Lin. Understanding the characteristics of android wear os. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, 2016.

[168] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[169] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[170] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

[171] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A Security Analysis of an In-Vehicle Infotainment and App Platform. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, WOOT'16, 2016.

[172] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[173] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[174] T. Nighswander, B. Ledvina, J. Diamond, R. Brumley, and D. Brumley. GPS Software Attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[175] A. Nikravesh, D. K. Hong, Q. A. Chen, H. V. Madhyastha, and Z. M. Mao. QoE Inference Without Application Control. In *Proceedings of the 2016 Workshop on*

QoE-based Analysis and Management of Data Communication Networks, Internet-QoE '16, 2016.

[176] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, 2013.

[177] S. Pernsteiner, C. Loncaric, E. Torlak, Z. Tatlock, X. Wang, M. D. Ernst, and J. Jacky. Investigating Safety of a Radiotherapy Machine using System Models with Pluggable Checkers. In *Proceedings of the 28th International Conference on Computer Aided Verification*, CAV '16, 2016.

[178] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl. Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR. In *Black Hat Europe*, 2015.

[179] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl. Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR. In *Black Hat Europe*, 2015.

[180] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: A cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, 2011.

[181] A. Rahmati and H. V. Madhyastha. Context-Specific Access Control: Conforming Permissions With User Expectations. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, 2015.

[182] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[183] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[184] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.

[185] F. B. Schneider. Least Privilege and More. In *IEEE Security & Privacy*, 2003.

[186] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a Formal Model of Safe and Scalable Self-driving Cars. *CoRR*, 2017.

[187] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium*, NDSS '16, 2016.

[188] H. Shin, D. Kim, Y. Kwon, and Y. Kim. Illusion and Dazzle: Adversarial Optical Channel Exploits Against Lidars for Automotive Applications. In *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'17, 2017.

[189] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava. Non-invasive Spoofing Attacks for Anti-lock Braking Systems. In *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'13, 2013.

[190] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information systems security*, 2008.

[191] R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *Proceeding Sof the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, 2008.

[192] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, 2008.

[193] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS'14, 2014.

[194] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9*, HICSS '02, 2002.

[195] Y. Wang and A. Rountev. Profiling the Responsiveness of Android Applications via Automated Resource Amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, 2016.

[196] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, 2008.

[197] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[198] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, MobiCom '12, 2012.

[199] R. White, H. I. Christensen, and M. Quigley. SROS: Securing ROS over the wire, in the graph, and through the kernel. *CoRR*, 2016.

[200] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, 2015.

[201] Y. Xie and A. Aiken. Saturn: A Scalable Framework for Error Detection using Boolean Satisfiability. In *TOPLAS*, 2007.

[202] Y. Xie and A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Trans. Program. Lang. Syst.*, 2007.

[203] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, SEC '12, 2012.

[204] C. Yan, W. Xu, and J. Liu. Can You Trust Autonomous Vehicles: Contactless Attacks against Sensors of Self-driving Vehicle. In *DEF CON 24*, 2016.

[205] S. Yang, D. Yan, and A. Rountev. Testing for Poor Responsiveness in Android Applications. In *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems*, MOBS '13, 2013.

[206] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance. In *Proc. of CODES+ISSS*, 2013.