Assuring Data Authenticity While Preserving User Choice in Mobile Sensing

by

Peter Gilbert

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Landon P. Cox, Supervisor

_____
Jeffrey S. Chase

_____
Jun Yang

_____
Romit Roy Choudhury

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2018

Abstract

Assuring Data Authenticity While Preserving User Choice in Mobile Sensing

by

Peter Gilbert

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Landon P. Cox, Supervisor

_____
Jeffrey S. Chase

_____
Jun Yang

_____
Romit Roy Choudhury

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2018

Abstract

As more services have come to rely on sensor data such as photos and audio collected by mobile phone users, verifying the authenticity of this data has become critical for service correctness. At the same time, contributors require the flexibility to modify data for resource efficiency, presentation, or privacy before the data is submitted. This dissertation presents two approaches for resolving the tension between data authenticity and user choice. YouProve is a partnership between a mobile device's trusted hardware and software that allows untrusted client applications to directly control the fidelity of data and enables services to verify that the meaning of source data is preserved. The key to YouProve's approach is trusted analysis of derived data, which generates statements comparing the content of a derived data item to its source.

To address certain cases where YouProve's approach is insufficient for evaluating modifications to photos, we introduce an alternative approach called pixel tracking. Pixel tracking uses dynamic taint analysis, or taint tracking, to monitor the execution of untrusted image processing code and track the history of operations performed on individual pixels. Pixel tracking is built on TaintDroid, a collaborative work that enables taint tracking in the Android operating system. This dissertation presents two key enhancements to TaintDroid to improve its efficiency and precision which are critical for enabling pixel tracking and other follow-on work.

Experiments with prototype implementations of YouProve and pixel tracking for

Android demonstrate that the approaches are feasible. YouProve's photo analyzer is over 99% accurate at identifying regions changed only through meaning-preserving modifications such as cropping, compression, and scaling. Pixel tracking complements YouProve's analysis and can provide valuable information in several important cases where the photo analyzer falls short. YouProve's audio analyzer is similarly accurate at detecting which sub-clips of a source audio clip are present in a derived version, even in the face of compression, normalization, splicing, and other modifications. Finally, performance and power costs are reasonable, with YouProve's analyzers having little noticeable effect on interactive applications and CPU-intensive analysis completing asynchronously in under 30 seconds for 5-megapixel photos and under 70 seconds for 5-minute audio clips. Pixel tracking incurs slowdowns of only 21% to 43% for fine-grained tracking of image processing code.

Our work on YouProve and pixel tracking demonstrates that it is possible to provide guarantees about data authenticity while preserving users' control over the data they contribute.

To the loves of my life, Katherine and Eliza.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Abbreviations

DEX	Dalvik Executable, Android's custom bytecode format.
JIT	Just-in-time compilation, an approach for executing bytecode that compiles bytecode to machine code at run time.
SSD	Sum of squared differences, a metric for approximating the visual similarity of two equal-sized blocks of image content. It computes the difference between the value of a pixel in the first image and the corresponding pixel in the second image, and then sums the square of these differences over all pixels in the block.
PPSSD	Per-pixel sum of squared differences, a block size-independent version of SSD, defined as the SSD value for a block divided by its area in pixels.
SURF	Speeded-up robust features, a local feature detector and descriptor in computer vision.
TCB	Trusted computing base, the set of all software and hardware components critical to the security of a system.
TPM	Trusted Platform Module, a hardware component with cryptographic capabilities that can enable a system to *attest* to its software configuration and *bind* statements to that configuration.
VM	Virtual machine, an emulation of a physical machine in software. Can refer to system VMs like VMWare or Xen, or process VMs like Android's Dalvik VM.

Acknowledgements

First, I would like to thank my advisor Landon Cox for his guidance, support and patience during my journey toward this milestone. I am grateful for the example that Landon set of pursuing exciting, challenging, and impactful research goals.

I would also like to thank my committee, Jeff Chase, Romit Roy Choudhury, and Jun Yang. It is an honor to have such accomplished researchers on my committee.

I am fortunate to have had the opportunity to work with many outstanding collaborators. I am especially grateful to Henry Qin and Jason Lee for their work on YouProve. It was a great pleasure to work with Will Enck, Jaeyeon Jung, Byung-Gon Chun, and Anmol Sheth on the TaintDroid project. I also enjoyed collaborations with Hamed Soroush, Nilanjan Banerjee, Brian Levine, and Mark Corner at UMass, as well as Doug Terry, Rama Ramasubramanian, and Patrick Stuedi during my internship at Microsoft Research Center.

I would like to thank my labmates, especially Eduardo Cuervo, whose contagious optimism and exuberance helped make my time at Duke so much more enjoyable. I would also like to thank Pablo Gainza for sharing so many afternoon coffees.

I am grateful to have worked with Dawn Song and Raymond Wei and my colleagues at Ensighta and FireEye, especially the original team of Prashanth Mohan, Adrian Mettler, Noah Johnson, Hui Xue, and Jimmy Su.

Finally, thank you to my family for supporting me on this journey. It would not have been possible without your love and support.

1

Introduction

Mobile phones are fast becoming the eyes and ears of the Internet by embedding digital communication, computation, and sensing within the activities of daily life. The next generation of Internet platforms promises to support services like citizen journalism, mobile social networking [37], environmental monitoring [54], and traffic monitoring [44] by pairing the ubiquitous sensing provided by mobile phones with the large-scale data collection and dissemination capabilities of the cloud.

Data authenticity is crucial for service correctness. Mobile social services have already been gamed by participants claiming to be in places they were not [45], and falsified images of natural disaster sites [7] and political protests [1, 4] have recently spread virally across social media, at times being re-broadcast by highly trusted traditional media sources. The emerging "fake news" phenomenon has demonstrated the devastating effectiveness of falsified online content at manipulating public opinion and unfairly harming political opponents. Thus, given the increasingly large role crowd-sourced content plays in world affairs and the dire consequences that dissemination of falsified media can have, verifying the authenticity of this data is paramount.

One proposed solution is to equip phones with trustworthy sensors capable of signing their readings and to require clients to return unmodified signed data to a service [33]. Unfortunately, requiring clients to send unmodified data is impractical. Mobile clients require the flexibility to trade-off data fidelity for efficient resource usage and greater privacy. For example, a client may wish to upload a photo with reduced resolution or under lossy compression to improve energy-efficiency and performance [19], or a client may wish to blur faces in a photo to conceal someone's identity [58].

Privacy is particularly important when mobile phones are used to collect and submit sensor data. Mobile phones store a wealth of personally identifying information such as phone numbers and device IDs, as well as the precise geographical location of the device. These types of sensitive information are often accessed by third-party applications, with users having little visibility into how the data is used or whether it is protected from exposure. An example involving an extremely popular app and a spyware product used by a number of repressive regimes illustrates the severity of the risks posed when apps mishandle sensitive data. WhatsApp, one of the most popular messaging platforms in the world, was found to have a critical flaw in its Android app that exposed the device owner's complete chat history to other apps installed on the device [12]. Later, the spyware product FinFisher, notably used to spy on activists countries like Egypt, Bahrain and Uganda, was discovered to be harvesting WhatsApp logs from infected devices [13]. Contributions from citizen journalists are particularly valuable in places where government bans and reprisals against journalists make traditional reporting difficult. In these cases, protecting the anonymity of a contributor could be a matter of life or death. To realize the full potential of user-contributed data for mobile sensing services, it is critical for users to have control over the privacy-sensitive information that might be revealed through their participation.

In this dissertation, I explore the tension between the need for content consumers to verify data authenticity and the need for contributors to have full control over the data they submit and the information revealed through their participation. I propose several approaches for making mobile sensing systems more trustworthy for both groups.

My thesis can be stated as follows:

> **It is possible to provide guarantees about the authenticity of data contributed by mobile users while preserving users' freedom to edit data for the purpose of privacy, efficiency, or presentation.**

The rest of the dissertation is organized into five chapters as follows:

Chapter 2 describes YouProve, a system that leverages trusted hardware and software on mobile devices to make sensing more trustworthy. It enables data contributors to directly control the fidelity of photo and audio data they upload and services to verify that the meaning of source data is preserved.

Chapter 3 provides an overview of TaintDroid, a collaborative work that enables dynamic tracking of information flows on the Android platform through system-wide dynamic taint tracking and analysis. This capability is useful for monitoring how third-party apps handle users' sensitive data, as well as for auditing how sensor data is processed before being uploaded to a mobile sensing service. Two significant enhancements to the original TaintDroid system are presented as contributions in this dissertation: (1) enabling taint tracking in the production version of Android's Dalvik VM, resulting in a 4x improvement in runtime performance, making it only 32% slower than unmodified Android, and (2) implementing fine-grained tracking for arrays, enabling systems requiring this fine-grained tracking to be built on the taint tracking platform.

Chapter 4 describes an alternative approach for monitoring how apps modify

photo data and ultimately determining whether a photo has been altered in a way that changes its meaning. The approach is specifically designed to handle cases that are difficult to evaluate using YouProve, e.g., detecting modifications to fine detail in the presence of scaling. This approach builds on the taint tracking capabilities provided by TaintDroid, including fine-grained tracking for arrays.

Chapter 5 describes related work, and **Chapter 6** concludes the dissertation with a summary of the key contributions.

2

YouProve: Authenticity and Fidelity in Mobile Sensing

This chapter presents YouProve, a system that aims to resolve the tension between data authenticity and user choice through a partnership between a mobile device's trusted hardware and software. It allows untrusted client applications to directly control the fidelity of photo and audio data they upload and services to verify that the meaning of source data is preserved.

2.1 Introduction

Many services have enlisted smartphone users to enable highly-scalable, low-cost sensing across a variety of domains. Election monitoring [6], citizen journalism, traffic monitoring [44], mobile social networking [37], and environmental monitoring [54] are just a few contexts in which useful sensing can be performed by inexpensive consumer devices.

However, services can realize the full potential of crowd-sourcing only when they can ensure that contributed data is authentic. Citizen journalism services such as Al

Jazeera's Sharek and CNN's iReport rely heavily on contributions from mobile users. Deploying trusted reporters and photographers into conflict zones or other unstable situations is difficult. Due to logistical obstacles, government bans, and reprisals against journalists, anonymous local citizens with camera phones have been instrumental in documenting important events taking place around the world. However, citizen journalism services have been fooled by falsified images [49, 70]. Given the increasingly large role crowd-sourced content plays in world affairs and the dire consequences that dissemination of falsified media could have, verifying the authenticity of this data is paramount.

A straightforward approach for assuring data authenticity is to equip phones with trustworthy sensors capable of signing their readings and to require users to submit unmodified signed data to a service [33]. Unfortunately, in many cases it is impractical for clients to send unmodified data. Mobile clients require the flexibility to trade-off *data fidelity* for efficient resource usage and greater privacy. This is particularly true for media such as audio and photos. It may be necessary to reduce the resolution or compress a photo before uploading for energy-efficiency and performance [19]. A user may wish to obscure faces in a photo to conceal the identities of bystanders [58]. Resolving the tension between data authenticity and data fidelity is a key obstacle to realizing the vision of phone-based distributed sensing.

Trusted hardware such as a Trusted Platform Module (TPM) can serve as the foundation of a solution. A partnership between a device's trustworthy hardware and its system software can produce digitally-signed statements about a data item's "chain of custody" to a remote service. However, even given this outline of a solution, several questions remain: What form should the partnership between device hardware and software take? What statements should a client present to a service? On what bases should a service trust a client's statements? What are the energy and performance implications of generating those statements on a resource-constrained

mobile client?

In this chapter, we address these questions by presenting the design and implementation of *YouProve*, a framework for verifying how data is captured and modified on a sensor-equipped mobile phone. We believe that the ability to verify that a derived data item preserves the meaning of an original sensor reading is an important step for evaluating data authenticity in domains such as citizen journalism. The key to our approach is *type-specific analysis* of derived sensor data. Type-specific analysis can be implemented by using well-known audio-analysis and computer-vision libraries to compare the *content* of a source item (e.g., an original audio clip or photo) to the content of a derived version of the item. The goal of type-specific analysis is to allow client applications to apply fidelity-reducing modifications to data and to give services a basis for trusting that those modifications preserved the meaning of the source.

To meet this goal, YouProve logs sensor data as it is requested by an application, and uses TaintDroid [34] to tag and track data as it flows through the application. If the application generates an output that is derived from a logged source, YouProve invokes the appropriate analyzer to compare the output to its source. The result of type-specific analysis is a *fidelity certificate*, which summarizes the software configuration of a device as well as how closely the content of a derived data item matches its source. By providing trustworthy statements about the degree to which a derived item's content matches its source, YouProve allows a service to verify that the meaning of the source item was preserved without requiring (1) clients to send the source, or (2) services to trust the applications that generated the derived item.

We have implemented a YouProve prototype for Android on a Nexus One and evaluated the performance and accuracy of audio and photo analyzers. Our prototype photo analyzer is over 99% accurate at identifying regions changed only through meaning-preserving modifications such as cropping, compression, and scaling. Our

prototype audio analyzer is similarly accurate at detecting which sub-clips of a source audio clip are present in a derived version, even in the face of compression, normalization, splicing, and other modifications. Finally, the performance and power costs of YouProve are reasonable, with logging having little noticeable effect on interactive applications and CPU-intensive analysis completing asynchronously in under 70 seconds for 5-minute audio clips and under 30 seconds for 5-megapixel photos.

2.2 Design Considerations

Mobile sensing services (also called participatory sensing [22]) consist of servers that collect, aggregate, and disseminate geo-tagged sensor data such as audio and images from volunteer mobile clients. The case for fidelity-aware mobile clients is well established [19, 48, 36, 40, 59], while the case for verifying the authenticity of sensing data has been made more recently [33, 39, 50, 66, 67]. *YouProve* is a trustworthy sensing platform built on Android that allows a client to control the fidelity of data it submits and sensing services to verify that the meaning of source data is preserved across any modifications. In this section, we provide background information on key aspects of YouProve's design, including descriptions of (1) the relationship between data authenticity and fidelity, (2) Trusted Platform Modules (TPMs), and (3) our underlying design principles.

2.2.1 Authenticity, fidelity, and trust

Two crucial concerns for a system that allows data fidelity to be traded off for energy, privacy, or other considerations are (1) what code modifies a source data item, and (2) what code consumes the derived item. An important part of verifying the authenticity of a derived item is the process of certifying that its content preserves the meaning of its source. We further define the term "meaning" in the context of audio and photo data in Section 2.5.

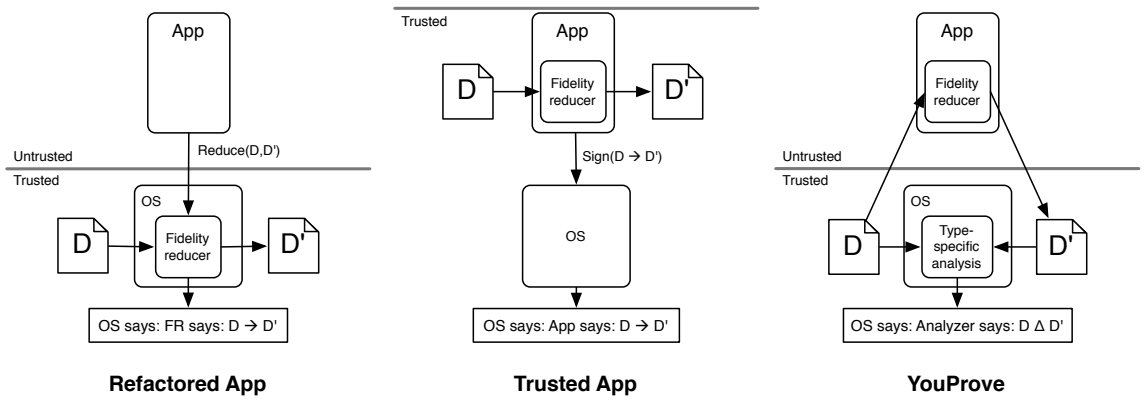

FIGURE 2.1: Possible locations of fidelity-reduction code.

Fidelity has traditionally been studied in the context of mobile clients retrieving data from servers over a wireless network [36, 48, 59]. In these settings, a small set of servers are trusted to maintain canonical copies of all source data and generate reduced-fidelity versions at the request of a client. Clients typically trust only a small set of servers based on the reputations of the servers' administrators. As long as a server can prove to a client that it is a member of the trusted set, the client considers the server's data to be authentic.

In a mobile sensing service, clients use sensors such as cameras, microphones, and GPS receivers to generate source data items and may produce a derived item by reducing the source's fidelity. Servers receive derived data and interpret its content to implement a service's logic. However, unlike in a traditional system like a distributed file system, a mobile sensing service cannot always rely on reputations to verify data authenticity. Data may be provided by clients without a prior history, by those who wish to remain anonymous, or by clients whose reputations are inaccurate due to Sybil-style gaming [32]. Other potential bases of trust are also problematic. Verifying authenticity by relying on a majority vote among related items is vulnerable to Sybil attacks, in which an attacker exercises disproportionate influence by creating a large number of identities. Relying on co-located trusted "witnesses" limits authenticity

guarantees to data from locations with trustworthy infrastructure [50, 66].

Several groups have sought to decouple client reputations from data authenticity using trusted hardware such as a Trusted Platform Module (TPM) [33, 39, 56, 67, 81] or ARM TrustZone [65]. TPMs are included in most PCs sold today, and a specification for a Mobile Trusted Module (MTM) for mobile phones has been released [18]. Similarly, most shipping ARM processors support TrustZone, which is a hardware-isolated, secure-execution environment [17]. A recent work leveraged TrustZone to implement TPM functionality in software on mobile devices [63]. We take the presence of trusted hardware on mobile clients as a given, and we have designed YouProve as a set of software services running on top of such hardware.

2.2.2 TPM background

TPMs provide a root of trust on each YouProve client. A TPM can be used to provide a verifiable boot sequence, in which each piece of code that runs during boot is *measured* by the cryptographic hash of its content prior to being executed. Each measurement is *extended* into a Platform Configuration Register (PCR) by the TPM such that the value of a PCR is loaded with a hash of its current value concatenated with the new measurement. Each TPM includes an array of PCRs that can be updated only through the extend operation and reset only by rebooting the device.

A TPM can *attest* to the state of its PCRs by generating a *quote* that is signed with a private key. Each TPM contains a unique public-private key pair called an Endorsement Key (EK) that is installed at manufacture time. Each EK uniquely identifies an individual device. To perform anonymous quoting, a TPM can generate new public-private key pairs called Attestation Identity Keys (AIKs). In order for services to trust an AIK, a trusted third-party privacy certificate authority (privacy CA) must generate a certificate for the public half of the AIK. We assume that trusted privacy CAs will only certify a small number of AIKs for each EK to limit

the scope of Sybil attacks.

A service can verify that a device's system software is trusted by checking that the PCR values reported in a quote match known values for a trusted configuration. A client's software configuration is trusted by a service if it ensures a trustworthy chain of custody for sensor data. As many have noted [56, 76, 81], verifying that the software platform of a mobile phone matches a trusted configuation is a promising technique because phone manufacturers release a relatively small number of read-only firmware updates that encapsulate the entire trusted codebase (TCB) of a device. Third-party applications are typically excluded from the TCB via an isolated execution environment. Furthermore, mobile devices do not give users root access by default, limiting opportunities for modifying the TCB configuation. This results in a high degree of homogeneity among TCB configurations on mobile devices. In contrast, traditional PC systems are more amenable to customization, resulting in numerous possible TCB configurations that must be evaluated.

While a user can gain root access on her phone by flashing a customized firmware with relative ease, such changes would be exposed to a remote verifier by a TPM's measurement of the boot sequence. Furthermore, users who root their phones are in the minority, allowing a service to reason about the trustworthiness of the vast majority of devices by establishing trust in a manageable number of configurations.

2.2.3 Design principles

The trusted chain of custody for sensor data must minimally include a trustworthy bootloader, OS kernel or hypervisor, and device drivers. There are a number of tradeoffs to consider in choosing the bases of trust for the rest of the chain. In particular, the critical challenge for YouProve is handling the code that performs fidelity reductions on source data. With this challenge in mind, we designed YouProve using the following principles:

Build on deployed systems.

We could have taken a clean-slate approach to YouProve by developing a new operating system or by using an experimental OS for trusted hardware like Nexus [68]. For example, alternative architectures using virtual machines provide a smaller trusted computing base than YouProve by removing components inessential to handling sensor data [39, 41, 67]. Instead we designed YouProve as a new set of trusted services for Android. Building on top of Android is simple, creates a lower barrier to deployment, and allows us to take advantage of existing Android tools. The disadvantage of our approach is that YouProve must be secured within Android's security model. We describe the process of securing YouProve in greater detail in Section 2.4.

Allow applications to directly modify data.

Verifying a data item's authenticity involves proving that it was derived from source data in a trustworthy way. One way to enable verification is to refactor applications by requiring trusted code to reduce the fidelity of source data on applications' behalf. Fidelity-reducing code would be included in the kernel or run as a trusted server in user space, and would use the trusted hardware to generate statements describing the reductions it performed. This approach is shown as Refactored App in Figure 2.1. The appeal of this approach is that it simplifies verification by limiting the number fidelity-reducing codebases that a service needs to trust.

APIs for controlling some forms of data fidelity at *capture time* already exist. For example, Android apps can adjust the fidelity of the location readings it receives via the `location.Criteria` class, the resolution and quality of the photos they receive via the `Camera.Parameters` class, and the sampling rate of audio via the `MediaRecorder` class. However, extending these existing APIs to support (1) a broader range of fidelity-reducing operations (e.g., cropping or blurring subregions of an image) and (2) data modifications during *post-processing*, would require significant

modifications to thousands of apps.

For example, there are nearly 6,000 apps under the "Photography" category of Apple's App Store, and many are media editors that operate on data captured in the past. Camera+, iMovie, and Garageband are several high profile editors. Similarly, Adobe Photoshop Express for Android has been installed over one million times. Furthermore, stand-alone media editors are not the only apps that perform fidelity reduction on data after it has been captured. Images taken using an iPhone 4 have a resolution of 2592x1936 pixels, but the Facebook API documentation strongly recommends that third-party apps resize images to a maximum of 2048 pixels along the longest edge before uploading [60]. Similarly, Instagram, one of the most popular photo-sharing services in the world, requires users to scale and crop images using the app to fit within a 1080x1080 pixel square before uploading [14].

As a result, rather than force existing apps to be refactored, YouProve allows unmodified third-party apps to continue to directly perform fidelity reduction at any point in a data item's lifetime.

Trust analysis rather than synthesis.

Another option is to allow apps to perform fidelity reduction themselves, but to provide a system API for generating signed statements about the app and its execution. This approach is exemplified by CertiPics, a trustworthy image-editing application developed for Nexus [68]. As long as a service trusts a program to correctly modify source data and to correctly describe what it did, then the service can verify the authenticity of the program's output. This approach is shown as Trusted App in Figure 2.1.

The primary disadvantage of this approach is that either (1) a user must restrict herself to using the small number of apps that her services trust, or (2) a service must establish trust in each of the thousands of apps a user might wish to use. Forcing

clients to use a small number of apps deemed trustworthy by her services undermines the surge of development activity that has made consumer mobile devices popular and useful. On the other hand, forcing services to reason about the trustworthiness of the thousands of apps that directly manage data fidelity is impractical. Mobile apps are generally closed source (making them difficult to inspect), and verifying the trustworthiness of tens of thousands of mobile developers is infeasible.

In the terminology of the Nexus Authentication Logic, both refactoring apps and certifying statements generated by trusted apps offer *synthetic bases* of trust. In both approaches, trusted code transforms source data and generates a signed statement about its output. This is similar to a trusted compiler generating a signed statement about the type safety of executable code that it outputs. However, as we have observed, synthetic bases of trust force developers to make significant modifications to existing apps, constrain which apps a user can use, or impose an impractical trust-management burden on services.

As a result, YouProve eschews synthetic bases of trust in favor of *analytic bases* of trust. An analytic basis for trust requires verifiers to trust code to analyze inputs as opposed to trusting code to synthesize an output. Trusted analyzers can be included in a device's firmware and allow untrusted apps to handle fidelity reduction. At a high level, an analyzer generates a report comparing the content of an application's output to the content of the original source. YouProve can then embed the analyzer's report and a measurement of the device's software configuration in a signed *fidelity certificate*. Services use fidelity certificates to reason about both the trustworthiness of a device and whether a derived data item preserves the meaning of its source.

Relying on trusted analysis rather than trusted synthesis allows YouProve to (1) simplify verification by placing all trusted code within the device firmware, (2) preserve the autonomy of apps to directly perform fidelity reduction, and (3) maximize users' choice of applications. This approach is shown as YouProve in Figure 2.1.

2.3 Trust and Threat Model

In this section, we discuss assumptions made by YouProve about a device's hardware and software configuration.

2.3.1 Trust assumptions

As stated in Section 2.2.2, the root of trust for each YouProve client is a TPM. If the private half of a TPM's EK or the private half of an AIK becomes compromised through an attack against the TPM hardware, then an attacker can generate arbitrary TPM quotes. However, as long as the privacy CAs that certify AIKs are not compromised, then an attacker will only be able to generate arbitrary quotes using the limited number of AIKs certified by the privacy CAs. Thus, as long as privacy CAs remain trustworthy, a single attacker cannot successfully masquerade as a large number of devices unless it compromises a large number of TPMs.

Building YouProve on top of Android instead of an experimental operating system was an explicit design choice, and a consequence of this choice is that YouProve inherits Android's security model. Thus, YouProve relies on Android's existing mechanisms to thwart attacks against the platform. The essential components of Android's security model are a Linux kernel, user-space daemons called services running under privileged UIDs, and an IPC framework called Binder. Each Android app is signed by its developer and runs as a Linux process with its own unique, unprivileged UID. Apps access protected resources such as the camera service through library code that makes IPC calls. Platform services limit interactions with untrusted application code by applying UID-based access-control policies on Binder communication.

YouProve assumes that modified firmware allowing users to execute code as root can be detected by inspecting a device's TPM quotes. Stock Android firmware typically does not give users root access, although some users will reflash their device

to gain root access. As long as the bootloader remains uncompromised, any changes to the user-modifiable firmware, which includes the trusted software platform, will be apparent to a remote service via the TPM quote embedded in a fidelity certificate.

Eliminating all software vulnerabilities is beyond the scope of this work—attacks against specific vulnerabilities in the implementation of a secure platform such as Android are long-standing and serious problems that are unlikely to disappear anytime soon. If an attacker manages to gain root access through a runtime exploit without modifying a device's trusted firmware, then the attacker can generate false analyses of sensor data. Nonetheless, as we will describe in Section 2.4, our design applies the principle of least privilege to isolate the effects of attacks against specific system components such as the camera and audio services and the log of source sensor data.

Other side-channel attacks that take advantage of physical access are beyond the scope of this work. YouProve stores encryption keys in memory and is therefore susceptible to "cold boot" attacks, in which an attacker retrieves residual data values from RAM after a hard reboot. An attacker could also inject false sensor readings while avoiding detection by physically interposing on the bus used for inter-device communication inside the phone. Specific hardware support would be needed to prevent these attacks.

Finally, "analog" attacks such as staged photos, photos of photos, or forged GPS signals are beyond the scope of YouProve. Incorporating data from multiple sensors into authenticity analysis could be helpful in detecting photos of photos or forged GPS signals; for example, it may be possible to identify a forged GPS location by checking if a contemporaneous temperature reading has a compatible value. However, there is only so much that a computer system can do to ensure data authenticity. Approaches like YouProve that consider original sensor readings as the "root" of authenticity cannot ensure that the event captured in a photo or audio recording was not staged.

2.3.2 Privacy implications

Previous work on trustworthy mobile sensing has suggested that a fundamental tradeoff exists between data authenticity and user privacy [39]. However, YouProve seeks to enhance both data authenticity and privacy by allowing a user to provide verifiable proof that a data item is authentic, even after fidelity reductions have been applied locally to remove identifying information. Rather than relying on the identity or reputation of a data contributor as a basis for trust, services can instead verify properties of a device's underlying software and hardware configuration. Thus, YouProve allows a device owner to attest to the authenticity of sensor data without revealing her identity.

On the other hand, YouProve's design also introduces new potential privacy risks for users. When a user publishes a fidelity certificate, she exposes detailed information about the hardware and software configuration of her device. However, it is important to note that the decision to upload a fidelity certificate is an explicit choice that remains under the user's control. We see YouProve as an opt-in service for users who wish to prove that data they generate is authentic.

While YouProve supports anonymous submissions, a design choice mentioned in Section 2.2.3 affects the extent to which a YouProve user can remain anonymous: reuse of an AIK when generating multiple certificates. Because TPM quotes signed with the same AIK can be attributed to the same physical device, anonymity may be compromised when a YouProve client reuses an AIK for multiple submissions. As previously mentioned, a privacy CA will certify only a small number of AIKs for a given device to limit the scope of Sybil attacks, in which a single device masquerades as multiple devices submitting data. Thus, a tradeoff exists between preventing Sybil attacks and enabling individual users to generate many unlinkable certificates. As a compromise, YouProve allows several AIKs to be certified for a device and leaves to

the user the responsibility of managing multiple identities associated with different AIKs. The choice of how many AIKs to certify for a device should be based on the potential damage to service quality due to Sybil attacks—we expect that the degree to which Sybil attacks can be tolerated will vary for different mobile sensing domains. The task of assisting users in managing multiple identities to prevent de-anonymization is left for future work.

2.4 YouProve

YouProve consists of four trusted software components responsible for performing the following tasks:

- *Logging* sensor data returned by the Android platform in response to requests from apps,
- *Tracking* data derived from sensor readings as it is manipulated by untrusted third-party apps,
- *Analyzing* the content of a derived data item and its source reading, and
- *Attesting* to the results of content analysis and the integrity of the software platform.

The rest of this section describes the design of each of these components.

2.4.1 Design overview

When an Android app wants to access sensor data it submits a request via a platform API (e.g., `Camera.takePicture`). YouProve interposes on such requests, assigning each request a unique identifier and inserting the response data along with its identifier into a *secure log*. YouProve must then track this data as it is modified by untrusted apps until a final data item is uploaded to a service provider. Standard

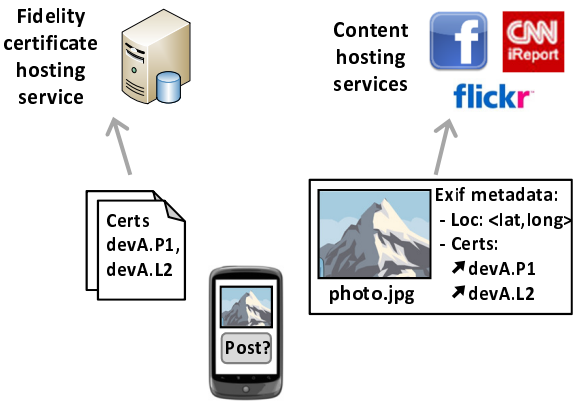

FIGURE 2.2: YouProve system overview.

meta-data tags are not sufficient for this purpose because sensor data may propagate through multiple file formats and representations. For example, consider an audio clip that is received by a third-party app as uncompressed samples in a memory buffer, processed by the app, converted to MP3 format, and then written to disk.

To track sensor data independently of its format, YouProve uses the TaintDroid [34] information-flow monitor to tag the response data with the identifier and propagate the tag to derived data in program memory, files, and IPC. TaintDroid is not used as a security mechanism. Rather, it is used to expose dependencies between source data and applications' outputs. The only consequence of TaintDroid's losing track of a taint tag would be that the authenticity of an application's output could not be verified with a fidelity certificate since the output would not be mappable to a source reading for analysis.

When an app generates an output such as a file write, YouProve inspects the output's taint tag. If the tag can be mapped to a logged data request, YouProve forwards the app's tagged output and its logged input to a *type-specific analyzer*. The analyzer synchronously generates an identifier for the output (e.g., by computing its cryptographic hash). At that point, YouProve can explicitly return the certificate identifier to the app or transparently embed it in the meta-data of the output such

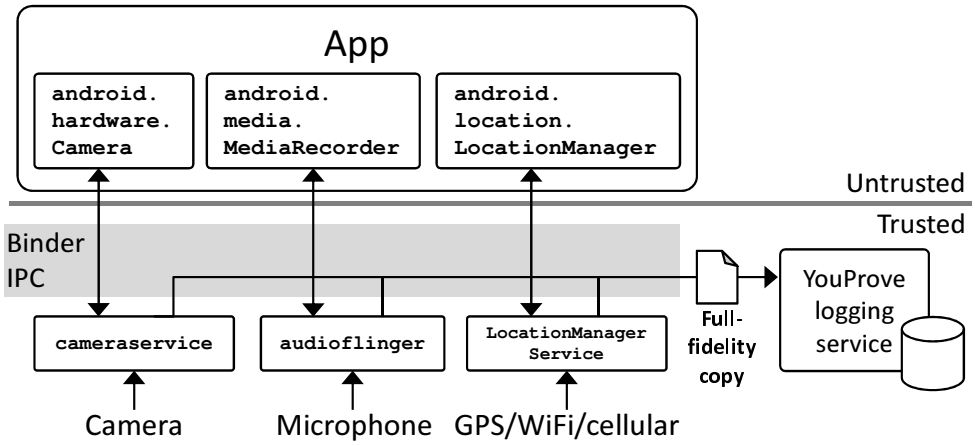

FIGURE 2.3: Logging sensor readings.

as an image's Exif data. The analyzer then asynchronously generates a report comparing the app's input and output. YouProve then embeds the report in a fidelity certificate and posts it to a remote hosting service. Figure 2.2 shows a high-level overview of this process.

A service that receives a data item can retrieve the item's fidelity certificate from a hosting service using the appropriate identifier, and can decide whether the data is authentic based on the content of the certificate.

This overview raises a number of questions: How does YouProve secure the log of responses and the type-specific analyzers? How do services establish trust in fidelity certificates? What information is included in a fidelity certificate?

2.4.2 Logging sensor readings

YouProve makes trustworthy statements about the content of a derived data item by comparing it to source data captured by a sensor. To support this analysis, it is necessary for the trusted platform to collect a full-fidelity copy of any sensor reading returned to an application and to protect the integrity of the stored copy as long as a user wishes to generate fidelity certificates for data derived from the reading. YouProve's *logging service* provides this functionality.

The logging service assigns a unique ID to each sensor reading and stores the original copy to a file. It also writes a database entry with a timestamp and type-specific metadata, as well as a pointer to the original copy on the filesystem and a digest of its contents. The database is stored on internal flash storage, while the actual data contents are stored on external storage (i.e., an SD card) due to the limited amount of built-in storage available on many smartphones.

The accuracy of the timestamp recorded for an original sensor reading is a critical part of data authenticity for many services. Because YouProve's timestamps are generated using a device's local clock, we must ensure that the clock is synchronized with a trustworthy source and protected from tampering. For this purpose, YouProve synchronizes the device's local clock with a trusted time server at boot-time using authenticated NTP [15].

On a typical smartphone platform, sensor data passes through a number of software layers between a hardware sensor and application code, including device drivers and user-level platform code. Choosing the level at which to log full-fidelity source data has implications for the size of the TCB, the performance and storage overheads of logging, and portability to different hardware devices. To minimize performance and storage overheads and maximize portability, YouProve captures data at a high level in the software stack, immediately before it enters an untrusted app's address space. As a result, full-fidelity copies have the same format, including encoding and possibly compression, as the data received by an app, and all platform code that handles sensor data prior to handoff to the app is included in the TCB. We discuss this tradeoff further in Section 2.6.

Android provides Java interfaces to apps for accessing camera, microphone, and location data. This Java library code runs in an app's address space and communicates via IPC with a system service designated to handle the specific type of sensor data. Android only allows the system services `cameraservice` and `audioflinger`

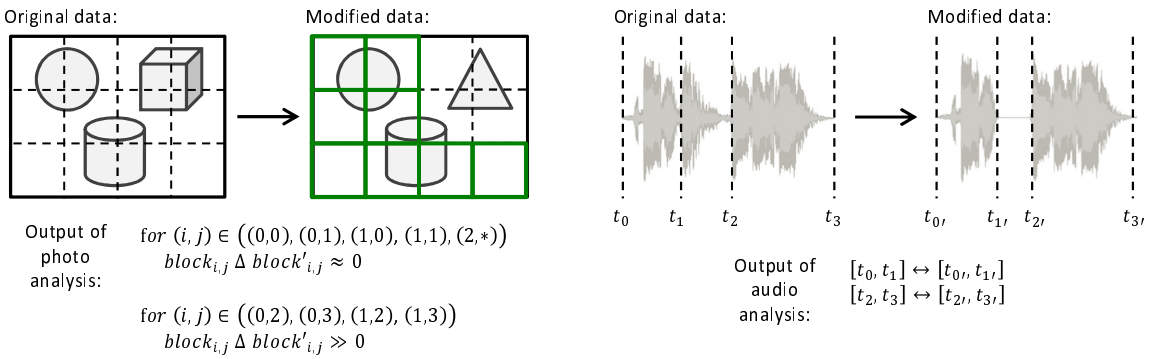

FIGURE 2.4: Overview of photo and audio content analysis.

to communicate with the camera and microphone device drivers, respectively. The `LocationManagerService` handles GPS and network-based location data. We instrumented these three system services to report sensor readings to the YouProve logging service via IPC before returning data to a requesting app. Figure 2.3 depicts the interaction between Android platform services and the YouProve logging service.

Ensuring the integrity of data recorded by the logging service is critical. If an attacker can impersonate one of the Android services entrusted to handle sensor data (i.e., `cameraservice`, `audioflinger`, or `LocationManagerService`) and submit inauthentic data to the logging service, fidelity reports will not be trustworthy. Trustworthiness will also be undermined if sensor readings or metadata in the log database can be modified by untrusted code without detection. As a result, YouProve must verify the identity of the services providing sensor data and verify the integrity of sensor data and metadata when it is read from persistent storage.

To authenticate requests to insert sensor data into the log, YouProve relies on Android's UID-based privilege-separation model and the process-identity information provided by Android's Binder IPC subsystem. Binder tells each endpoint of an IPC connection the UID of the other communicating process. A special UID "media" is assigned to the `mediaserver` process that hosts `cameraservice` and `audioflinger`. `LocationManagerService` runs under the UID "system", which is shared by a num-

ber of trusted services. Android executes only trusted system code under the "system" UID, and no process may run under the "media" UID after the `mediaserver` launches during Android's boot sequence. YouProve leverages the restrictions placed on these UIDs by the kernel to authenticate requests to log data. The logging service only accepts requests to log photo and audio data from processes with the "media" UID (i.e., the `mediaserver`) and requests to log location data from processes with the "system" UID (i.e., the `LocationManagerService` and other privileged services).

In addition to authenticating requests to the logging service, YouProve must also protect the integrity of the log database and the full-fidelity copies stored on disk. Full-fidelity copies are kept on external storage and may be vulnerable to external modification. YouProve ensures that modifications to these copies will be detected by verifying the SHA-1 digest of the data against the value stored in the log. To protect the integrity of the log database, YouProve signs each log entry with the private half of a key pair generated at install-time and bound to a trusted software configuration using a TPM's *sealed storage* functionality.

Sealed storage allows the system to submit arbitrary data to the TPM to be "sealed" to the current values of selected PCRs. Upon such a request, the TPM returns an encrypted blob that can later be "unsealed" only when the same PCRs are in the same state. This ensures that the private key used to sign log entries will be accessible to device software only after the trusted platform has booted into a known, trusted state. Note that YouProve does not attempt to prevent denial of service attacks wherein a user deletes full-fidelity copies or log entries—these attacks simply result in the user being unable to attest to fidelity-reduced data.

2.4.3 Tracking derived data

YouProve's type-specific analyzers operate on two data items: a full-fidelity source item and a derived version of the source. To enable comparisons YouProve relies on

the TaintDroid [34] information-flow monitoring framework as a lightweight means for tracking data dependencies throughout the Android system. Before the platform returns sensor data to a user app, it attaches a taint tag encoding the 32-bit unique ID assigned to the sensor reading—this ID serves as the primary key for the entry in the log database. If tainted data is appended to a file or IPC message already marked with a different ID, the file or message is marked with a newly allocated ID and the mapping to the two previous IDs is recorded in the log. In this way, YouProve can properly track all dependencies for high-level data items such as geo-tagged images. Due to space constraints, we have left out a longer discussion of handling data items whose taint tags map to multiple sensor readings. Our YouProve prototype currently supports geo-tagged camera and microphone data.

2.4.4 Analyzing content

YouProve's type-specific analyzers report the differences between an original sensor reading and a derived data item. Analyzers implement a simple, common interface: they take as input two files containing sensor data, and they output a human-readable report identifying portions of the modified data item that preserve "meaning" from the original sensor reading. Additional information may be provided about the differences between corresponding regions in the source and derived items when their contents do not match.

For both photo and audio data, YouProve's approach is to divide a derived data item into smaller regions and then attempt to match the content of each region to that of a corresponding region in the source sensor reading. Photos are divided by rectangular grid, while audio analysis considers time segments. An overview of YouProve's approach and the basic format of the output of analysis is shown in Figure 2.4. Our prototype analyzers for photo and audio content are described in detail in Section 2.5.

```
<cert dev_id="device pseudonym" cert_id="unique per device">
  <report>
    <content_digest>SHA1(content)</content_digest>
    <timestamp>from original sensor reading</timestamp>
    <analysis>type-specific content analysis results</analysis>
  </report>
  <report_digest>SHA1(report)</report_digest>
  <platform>
    <pcr0>
      <boot>SHA1(boot partition)</boot>
      <system>SHA1(system partition)</system>
    </pcr0>
    <aik_pub>AIK_pub</aik_pub>
    <tpm_quote>sig{PCR_0, report_digest}_AIK_priv</tpm_quote>
  </platform>
</cert>
```

FIGURE 2.5: Format of a fidelity certificate.

2.4.5 Attesting to analysis and platform

To enable data consumers to reason about the trustworthiness of a data item, YouProve's *attestation service* generates fidelity certificates that report the results of type-specific analysis and a timestamp for the original reading. Fidelity certificates also contain information about the device's software platform, allowing remote verifiers to decide whether or not to trust reports generated by the device. The format for fidelity certificates is shown in Figure 2.5. The two basic parts are (1) a *report* that describes a data item and is bound to the data by a content digest, and (2) a description of the platform software configuration, including a TPM quote that attests to the state of the software platform and binds the platform-specific part of the certificate to the content-analysis part.

Fidelity certificates are posted by a YouProve client to a remote *certificate hosting service*—a simple key-value store that makes submitted certificates available through well-known public URLs. YouProve generates fidelity certificates in response to

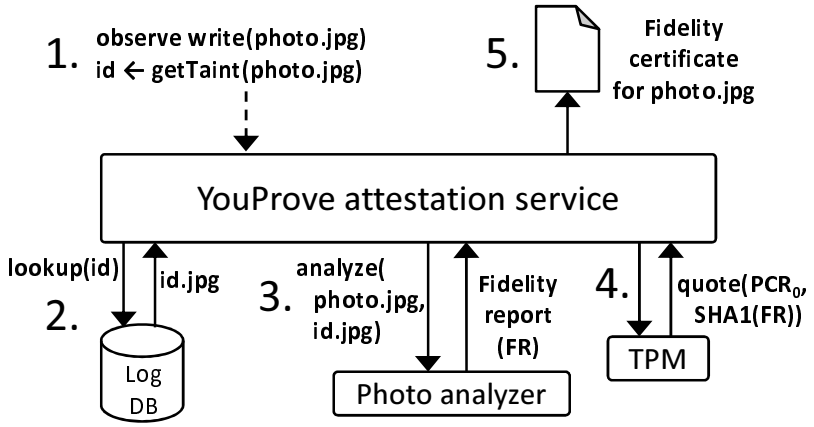

FIGURE 2.6: Steps for attestation.

explicit requests by the user, or by monitoring the user filesystem for writes to files with supported data types (e.g., jpegs and mp3s). If desired, a link to the URL where the certificate will eventually be available can be embedded as metadata in the media file before uploading to a sensing service. The basic steps performed by YouProve to generate a fidelity certificate are shown in Figure 2.6.

To verify a certificate, a service first verifies the signatures and hashes. Next it uses the analysis report to evaluate a service-defined policy regarding the authenticity of received data. We imagine that an analyzer's report will most commonly be used to strengthen the case for an item's authenticity. Items with ambiguous reports leave services in their current positions of relying strictly on other means to verify data's authenticity. We discuss the kind of information analyzers embed in their reports in the following section.

2.5 Type-specific Analyzers

In this section, we describe YouProve's analyzers for photo and audio content. The central design challenge was to apply state-of-the-art content analysis techniques without incurring excessive runtime or energy overheads.

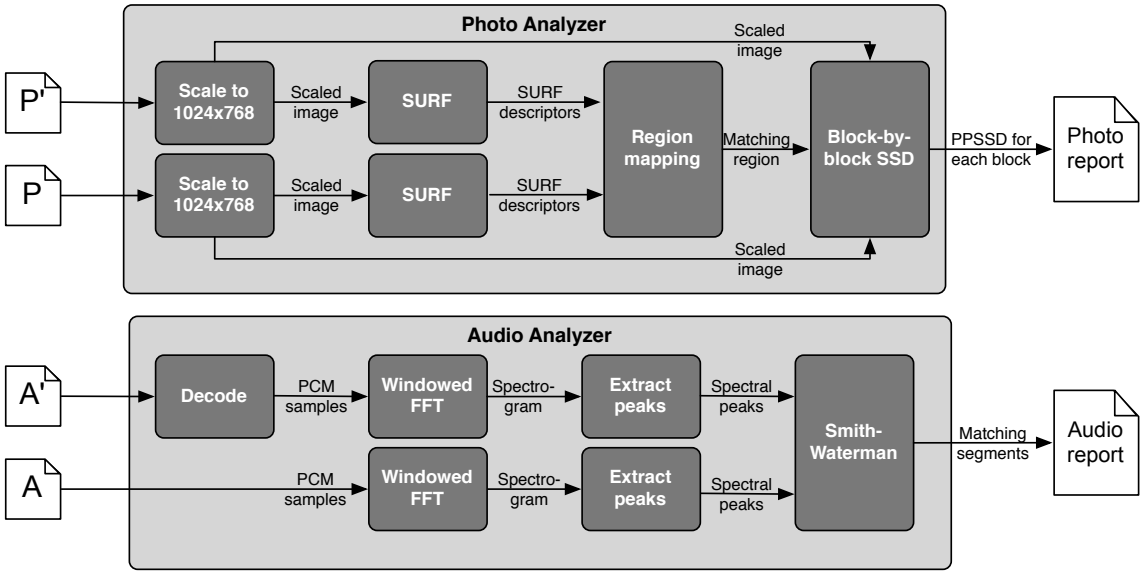

FIGURE 2.7: Photo and audio analyzer pipelines.

2.5.1 Photo content

The goal of YouProve's photo analysis is to identify regions of a derived photo that preserve meaning from a source photo captured by the camera hardware. We consider the meaning of a photo to be preserved if its appearance remains roughly unchanged. Thus, transformations that typically preserve the meaning of photo content include image compression, relatively small adjustments to image parameters such as brightness and contrast, and fixed-aspect ratio scaling. Furthermore, cropping preserves the appearance of any regions which are not removed. We consider manipulations that distort the image or significantly change regions in other ways (e.g., pasting in content from another photo) to alter the meaning of the sub-region of the photo. Ultimately, we seek to allow a viewer to categorize sub-regions of a modified photo based on whether they preserve meaning from a corresponding region in the source photo.

Our analysis utilizes two well-known techniques from computer vision: Speeded-Up Robust Features [20] (SURF) and Sum of Squared Differences (SSD). SURF

facilitates matching regions in two images by locating "keypoints" such as corners, and then computing a feature vector called a *descriptor* to represent each keypoint. A "matching" between the descriptors in two images can be found by computing the distance between the vectors. SSD is useful for approximating the visual similarity of two equal-sized blocks of image content. It is a simple metric that computes the difference between the value of a pixel in the first image and the corresponding pixel in the second image, and then sums the square of these differences over all pixels in the block.

Analysis procedure

Before comparing a derived photo with its source, we scale down both photos as necessary so that each fits inside a 1024x768 pixel bounding box. The maximum resolution for the Nexus One camera is 2592x1944; thus, source photos will be scaled down by a factor of at most 2.5. This is necessary due to the memory requirements of our analysis routines and the limited memory available on our target mobile device (512MB RAM for a Nexus One). As device RAM increases we will not need to scale images as drastically.

In addition, all analysis operates on grayscale versions of the photos, produced by taking a weighted average of the RGB channels. We believe that scaling down source photos to 1024x768 and converting to grayscale should not hinder our ability to recognize preserved regions and localize transformations that alter the source's meaning. If color modifications not apparent in the grayscale version are a concern, we can perform the same analysis on each of the three RGB channels at the expected expense of a 3x increase in runtime.

After resizing the input photos, the analysis proceeds in two phases. In the first phase, the analyzer attempts to find a correspondence between the derived photo and a region in the source photo. Note that in some cases the entire modified photo

will map to a sub-region of the source photo due to cropping. To find this mapping, we use SURF to extract descriptors from each photo and then compute the set of matching descriptors. We then find the minimum-sized rectangular region (with sides parallel to the coordinate axes) in the source photo containing the keypoints of all matching features—this is the region that the analyzer considers for further investigation. At this point, we scale down the larger of the matching region and the derived photo to make their sizes equal. If no mapping is found between the content of the modified photo and the source photo, we continue analysis assuming that the entire source maps to the derived photo.

After identifying the matching region of the source photo the analyzer subdivides both the derived photo and the matching region into equal-sized, approximately-square blocks, with eight blocks along the longer dimension, subject to a minimum block size of 32x32 pixels. This heuristic is intended to localize content-altering modifications with sufficient precision, while ensuring a reasonable number of pixels for computing SSD. Once the images have been divided into blocks, we compute the SSD of corresponding blocks.

A caveat of SSD's pixel-by-pixel comparison is that it is highly sensitive to alignments of image regions off by even a single pixel. To account for potentially imprecise alignments found using SURF, for each block we take a sub-image from the center 12% smaller in each dimension and compare it to each equal-sized sub-image in the corresponding block from the other photo. For example, if the block size is 128x128 pixels, we take a 112x112-pixel sub-image from one block and compare it with each of the 256 possible equal-sized sub-images from the other block. We record the minimum of the computed SSD values for each pair of blocks.

To account for different block sizes, we define a block size-independent metric, *per-pixel sum of squared differences* (PPSSD), as the SSD value for a block divided by its area in pixels. In Section 2.7, we report PPSSD values resulting from various

modifications and show that it is possible to define a threshold on PPSSD values which segregates blocks of a photo preserving meaning from the source from those containing content-altering local modifications.

The entire pipeline of the photo analyzer is depicted in Figure 2.7.

2.5.2 Audio content

Similar to our photo analysis approach, the goal of YouProve's audio analysis is to identify contiguous time segments of audio which were modified only in ways that do not alter the way the audio will be perceived by a listener. Transformations that typically preserve the meaning of audio content include compression, slight changes to volume, and "enhancing" filters such as normalize. Other manipulations such as time distortion and splicing in audio from other sources are considered to alter the meaning of the audio clip.

At a high level, YouProve's audio analyzer extracts sequences of *spectral peak frequencies* from the source and derived audio clips and applies *local sequence alignment* to find matching time segments. The use of spectral peak analysis to compare audio data was inspired by the Shazam audio recognition system [75]. The technique is well-suited for our analysis because spectral peaks are a central feature in human hearing and thus are independent of audio encoding. They are largely maintained across transformations that preserve the way audio will be perceived (e.g., compression). To identify time segments in a modified clip which preserve sequences of spectral peak frequencies from a source clip, we use a modified version of the well-known Smith-Waterman algorithm [69] for local sequence alignment.

We note that the naive approach of simply performing sequence alignment on audio samples is unsuitable for two reasons. First, sample values are completely changed by even simple transformations such as normalization, which adjusts volume and therefore the value of all samples. Second, the sequence alignment algorithm

runs in time proportional to the product of the sequence lengths. Because there are generally hundreds to thousands of samples in a 0.1-second segment of audio (the interval over which we compute each spectral peak frequency value), the naive approach would be slower by a factor of 10^4 to 10^6 for sequence alignment.

Analysis procedure

Our analysis begins by verifying that the two input audio clips have the same sampling rate and audio format to ensure an equal number of samples for a given time duration. The `audioflinger` service, which supplies the full-fidelity source clip to YouProve's logging service, outputs raw PCM samples in WAV format. If the modified clip is stored in a compressed format, we first decompress it to raw PCM data. At this point, we proceed with analysis only if the sampling rates of the source and derived clips are equal. In practice, the sampling rate of a modified clip rarely differs from that of the source clip, because the Android audio recording API allows applications to request a particular sampling rate from the `audioflinger` service. YouProve's logging service captures the source clip at this requested sampling rate, and we assume that applications will not change the sampling rate of a clip once it has been captured.

After verifying the aforementioned assumptions, we continue by computing the frequencies of spectral peaks for each audio clip as follows. First, we compute N, the number of samples in 0.1 seconds of audio. For each region of N samples in the clip, we perform a windowed Fourier transform to convert the samples into a collection of frequency bins and their amplitudes. We then find the frequency bin with the greatest amplitude and output that frequency. The result is a sequence of frequencies for each clip.

To facilitate finding multiple matching time segments, we modify Smith-Waterman to make the gap penalty infinite and then examine local alignments with scores above

a certain threshold (which we set), rather than simply the max scoring (optimal) local alignment. To deal with limited floating-point precision and to allow for some variation due to compression, we consider two frequencies a match for scoring purposes if the frequencies are within 10 Hz of each other. This value was chosen because at a 44.1 kHz sampling rate, adjacent frequency bands roughly differ by 10 Hz.

Each local alignment maps a specific range of audio samples (technically 0.1-second length clusters of samples) in one audio clip to a specific range of samples in the second. We order these alignments by descending score, and add them, in order, to our output set of alignments as long as there is no overlap with pre-existing alignments in either of the sequences. For the final output, we convert ranges of samples to regions of time.

As long as the edits performed to produce a modified clip do not sub-divide the source clip at finer than 0.1-second granularity, we expect to be able to identify preserved segments from the source clip. Moreover, even if edits splice at finer than 0.1-second granularity, this should not cause us to falsely label time segments in the modified clip that do not actually preserve content from the source clip.

The entire pipeline of the audio analyzer is depicted in Figure 2.7.

2.6 Implementation

In this section, we describe our prototype implementation of YouProve for Nexus One smartphones and Android 2.2.

YouProve's logging and attestation services are implemented in approximately 2,000 lines of Java code. Like all other commodity smartphones of which we are aware, the Nexus One does not include a TPM chip. Instead, we ported a popular open-source TPM emulator [10] to Android, along with the TrouSerS open-source TCG stack [11]. Our prototype photo and audio analyzers are written in C++ and C, respectively. The audio analyzer uses the open-source LibXtract and FFTW

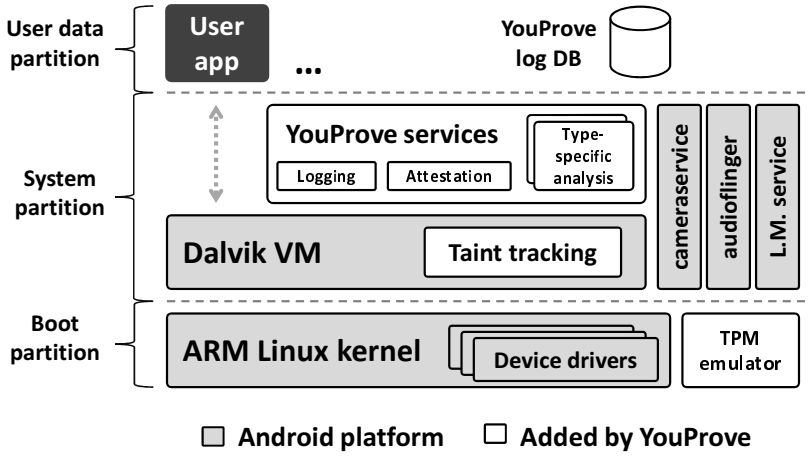

FIGURE 2.8: YouProve prototype architecture.

libraries for converting audio samples into frequency bins. We ported these libraries to the Android platform. The photo analyzer uses the SURF implementation from the popular open-source computer-vision library OpenCV. In addition, we use the open-source TaintDroid information-flow tracking framework [34], which we updated from Android 2.1 to 2.2.

As discussed in Section 2.2, YouProve builds upon the security model offered by the underlying Android platform. An important feature of this model is that any code that runs with elevated privileges must be loaded from the device's read-only firmware. Android partitions the device's internal flash storage into at least three partitions: the "boot" partition contains the Linux kernel and an initial ramdisk, the "system" partition includes all user-level Android platform code, and the "user data" partition stores the user's apps and data. The boot and system partitions comprise the read-only portions of the firmware. Effectively, all trusted code is loaded from these two partitions.

All trusted YouProve code is added to the read-only firmware. The arrangement of YouProve's software components and relevant Android platform components is shown in Figure 2.8. To enable an Android device to attest to the state of its

TCB, YouProve modifies Android's boot procedure to measure the boot and system partitions when they are mounted. Because Android's bootloader does not provide support for measured boot, our prototype actually measures both partitions after the boot partition is loaded by the bootloader and control has passed from the bootloader, to the kernel, and finally to the first user task, `init`. These measurements are extended into one of the TPM's PCRs to support subsequent attestation using TPM quotes.

The TPM quote included in a fidelity certificate attests only to the value of the bits of the firmware booted by the device. The trustworthiness of fidelity certificate contents depends on whether the booted firmware provides sufficient protection for the TCB.

2.7 Evaluation

In evaluating YouProve, we sought to answer two questions: (1) Can YouProve's type-specific analyzers accurately identify modified portions of a data item that preserve the meaning of its source? and (2) What are the performance and power costs of running YouProve?

2.7.1 Analyzer Accuracy

Current mobile sensing services have no way to verify the authenticity of the data they accept. YouProve's goal is to improve on this state by identifying regions of a derived data item that preserve the meaning of its source, while minimizing the number of incorrectly categorized regions.

Photo analysis

As described in Section 2.5.1, YouProve's photo analysis compares two photos block-by-block and reports a per-pixel SSD (PPSSD) value for each block of the derived

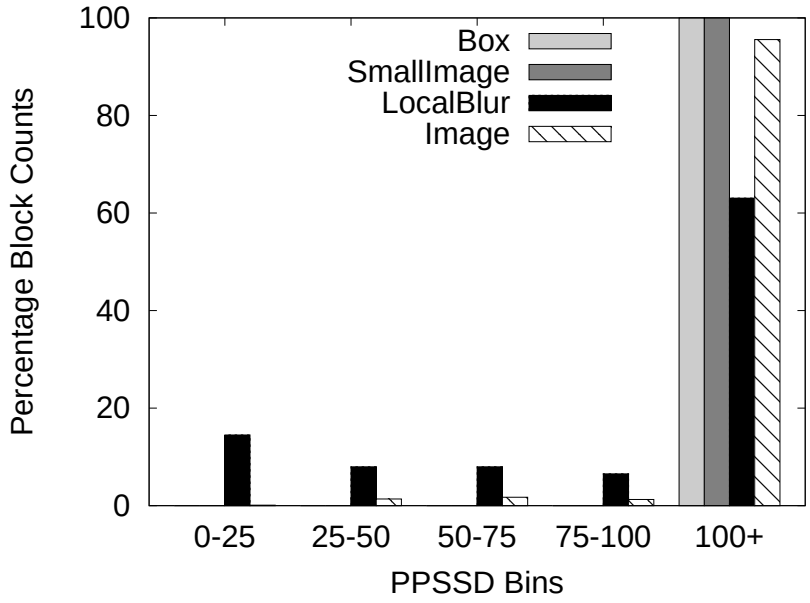

FIGURE 2.9: Block PPSSD for different local modifications.

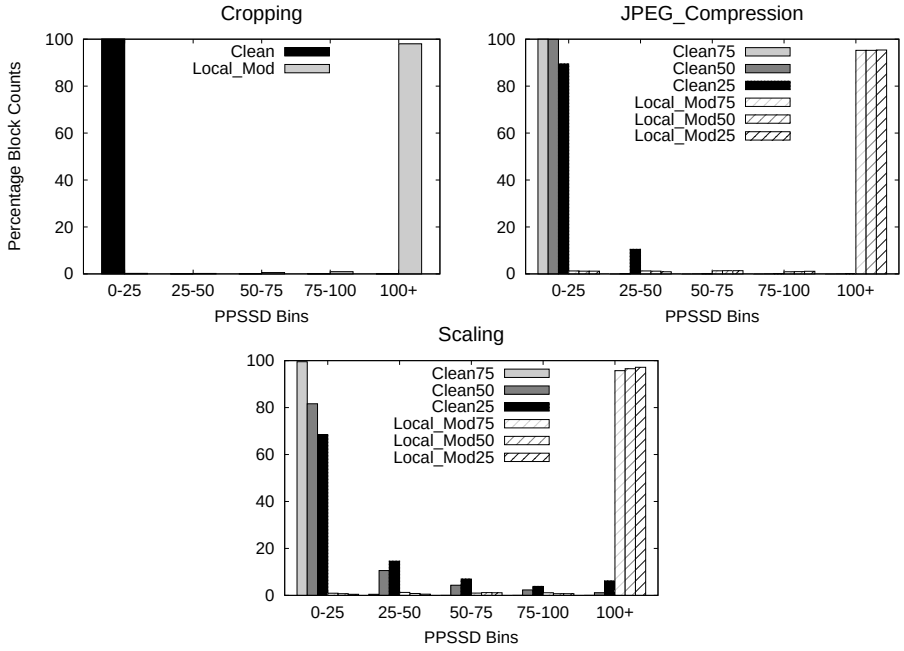

FIGURE 2.10: Results of photo analysis accuracy experiments.

photo. The first goal of our evaluation was to determine whether paired-block PPSSD values provide a good metric for identifying blocks that preserve the meaning of their source. We also sought a guideline PPSSD threshold for categorizing blocks.

The basis for our test dataset was a diverse collection of sixty-nine photos taken on a college campus using a Nexus One. Subject matter included individual students, crowds, buildings, offices, landscapes, walls with flyers, and bookshelves full of books. The photos varied in level of detail, level of focus, and quality of lighting. All photos were taken at the default resolution of 1944x2592 pixels. We then used the ImageMagick image-editing tool to apply two classes of modifications to our photos.

Global modifications included cropping, scaling, and JPEG compression. We consider these modifications to preserve the meaning of the source. Cropping test cases were created by cropping out either the top, bottom, left, or right half of an image, leaving a rectangular half-image. Scaling maintained aspect ratio while reducing each dimension to either 75%, 50%, or 25% of its original size. Compression produced JPEGs at 75%, 50%, and 25% quality.

Local modifications included overlaying a black box, pasting in a small photo, pasting in a large photo, or blurring a region. We consider these modifications to alter the meaning of the source. Our goal was to determine whether our photo analyzer could identify blocks containing local modifications, possibly even if global modifications had also been applied. To generate a test case containing a local modification, we applied one of the four types to a random, fixed location of the source. The black box, small photo, and blur regions were scaled with the dimensions of the photo to cover approximately 3% of the photo's area. The large photo covered approximately 30% of the area. Blocks that were untouched by a local modification were considered *clean*, even if they were transformed by a global modification.

As a basic test of the ability of YouProve's photo analysis to identify local modifications, we included in our first experiment only local modifications and no global

modifications. We ran photo analysis on these photos and measured the PPSSD of locally-modified blocks and clean blocks.

This experiment showed a clear separation between the PPSSD values of locally modified blocks and clean blocks: all clean blocks exhibited PPSSD values of less than 25, indicating a high degree of similarity to the source. However, slightly more than 3% of locally modified blocks also had PPSSD values of less than 50. To understand the cause of these low PPSSD values, we separated the PPSSD distribution of each local modification, as shown in Figure 2.9. Note that the y-axis of Figure 2.9 shows the percentage of blocks that were modified in a particular way, not the percentage of all locally-modified blocks.

The histogram shows that blocks containing blurred regions often exhibit low PPSSD values: over 15% of blurred blocks exhibited a PPSSD value of less than 25 and only 60% of blurred blocks exhibited a PPSSD value of greater than 100. The reason for these numbers is that blurring a region of nearly solid color produces very little change, and we did not bias our choice of region to blur based on its level of detail. Manual inspection of blurred blocks with low PPSSD values indicated that many arose from blurring already blurry regions, or blurring solid colors such as white office walls and the blue sky. In other words, the appearance of the blurred region was not significantly changed in these cases.

Next, we evaluated the accuracy of YouProve's photo analysis in the presence of global modifications. The analyzer's ability to handle cropping depends on the accuracy of our SURF-based approach for mapping a derived photo to a region of the source. Thus, we applied local modifications to a set of cropped photos and then compared the PPSSD values for clean blocks and locally-modified blocks within a cropped region. Figure 2.10 shows that all of the clean blocks within our cropped regions registered a PPSSD of less than 25. Furthermore, less than .5% of locally-modified blocks exhibited PPSSDs of less than 50. This demonstrates that SURF

Table 2.1: Miscategorized blocks, PPSSD threshold = 50.

Global modification	**Clean, PPSSD>50**	**L-Modified, PPSSD⩽50**
None	0.0%	3.10%
Cropping	0.0%	0.43%
JPEG quality: 75%	0.0%	2.50%
JPEG quality: 50%	0.0%	2.37%
JPEG quality: 25%	0.09%	2.11%
Scaling, 75%	0.0%	2.17%
Scaling, 50%	7.79%	1.52%
Scaling, 25%	16.94%	0.99%

is highly accurate in mapping a cropped photo to the corresponding region in the source.

It is interesting to note that the percentage of locally-modified blocks in cropped images with PPSSDs of less than 50 decreased by a factor of more than 7 compared to the case in which there were no global modifications. The reason for this is that cropping re-allocates blocks to the smaller area so that there are more blocks covering the same area of the image. For example, a face that was covered by one block before cropping might be covered by four blocks after cropping. In general, this will lead to a smaller proportion of "barely-changed" blocks, i.e., blocks with almost all preserved content and a small piece of locally-modified content. Because we only considered blocks that were completely untouched by local modifications to be clean, fewer blocks with only minor changes reduced the number of locally modified blocks with low PPSSD values.

Finally, we evaluated our photo analyzer's robustness to compression and scaling. To better understand how these transformations affect the PPSSD distribution of clean blocks, we investigated several degrees of scaling and compression. From Figure 2.9, we observe that the degree of compression has almost no impact on

Table 2.2: Audio analysis accuracy results (no additional compression).

Modification	**Music**		**Comedy**		**Speech**		**Lecture**	
	correct	*false*	*correct*	*false*	*correct*	*false*	*correct*	*false*
None	1.0	0.0	1.0	0.0	0.999	0.0	1.0	0.0
MP3, 128 kbit/s	0.999	0.0	0.956	0.0	0.996	0.0	0.949	0.0
MP3, 64 kbit/s	0.999	0.0	0.887	0.0	0.998	0.0	0.949	0.0
Dither	1.0	0.0	1.0	0.0	0.999	0.0	1.0	0.0
Double pad	1.0	0.0	1.0	0.0	0.833	0.0	1.0	0.0
Normalize	1.0	0.0	1.0	0.0	0.999	0.0	1.0	0.0
Replace w/ noise	-	0.066	-	0.0	-	0.0	-	0.0
Lower pitch	-	0.073	-	0.0	-	0.0	-	0.017
Raise pitch	-	0.054	-	0.0	-	0.0	-	0.0
Remove middle	1.0	0.0	1.0	0.0	0.999	0.0	1.0	0.0
Remove multiple	1.0	0.0	1.0	0.0	0.988	0.0	1.0	0.0
Segment splice	1.0	0.014	1.0	0.0	0.998	0.0	1.0	0.0
Crop	1.0	0.0	1.0	0.0	0.998	0.0	1.0	0.0

correct: proportion of preserved regions correctly identified
false: proportion of modified regions incorrectly identified as preserved

the PPSSD distributions. However, the PPSSD distribution for clean, scaled blocks shifts to the right as the degree of scaling increases. Luckily, the photo analyzer can include the scaling factor exhibited by a derived photo in its report so that services can increase their PPSSD threshold for more scaled-down images.

In light of our results, we believe that a threshold of 50 PPSSD is reasonable for identifying photo blocks that preserve the meaning of their source for photos that are not scaled down by more than 50%. The rates of miscategorization for a threshold of 50 PPSSD for all experiments are summarized in Table 2.1. For photos scaled down to 25%, a higher threshold of 100 PPSSD greatly improves our overall accuracy. Using this threshold, only 6.16% of clean blocks are miscategorized as modified, while 2.83% of locally modified blocks are miscategorized as clean.

We further observe that that the PPSSD accuracy loss we suffer in scaling test

Table 2.3: Audio analysis accuracy results (with compression).

Clip compressed and decompressed before applying modification

Modification	**Music**		**Comedy**		**Speech**		**Lecture**	
	correct	*false*	*correct*	*false*	*correct*	*false*	*correct*	*false*
None	1.0	0.0	1.0	0.0	0.999	0.0	1.0	0.0
MP3, 128 kbit/s	1.0	0.0	1.0	0.0	0.77	0.0	1.0	0.0
MP3, 64 kbit/s	1.0	0.0	1.0	0.0	0.77	0.0	1.0	0.0
Dither	0.999	0.0	0.956	0.0	0.996	0.0	0.949	0.0
Double pad	0.999	0.0	0.822	0.0	0.984	0.0	0.939	0.0
Normalize	0.999	0.0	0.956	0.0	0.996	0.0	0.949	0.0
Replace w/ noise	-	0.065	-	0.0	-	0.0	-	0.0
Lower pitch	-	0.057	-	0.0	-	0.0	-	0.02
Raise pitch	-	0.053	-	0.0	-	0.0	-	0.0
Remove middle	0.999	0.0	0.809	0.0	0.993	0.0	0.919	0.0
Remove multiple	0.999	0.0	0.753	0.0	0.992	0.0	0.9	0.0
Segment splice	0.999	0.013	0.762	0.0	0.984	0.0	0.903	0.0
Crop	0.98	0.0	0.58	0.0	0.784	0.0	0.982	0.0

correct: proportion of preserved regions correctly identified
false: proportion of modified regions incorrectly identified as preserved

cases does not appear to afflict us during the scaling phase of our analysis, as demonstrated by the low SSD values of the clean blocks in the other experiments. We attribute this to the fact that our photo analyzer scales both images with the same algorithm using the OpenCV library, while ImageMagick uses a different scaling algorithm.

Audio analysis

As described in Section 5.2, our audio analyzer compares two audio files, and reports time regions in one file that it determines to be derived from the other.

We aim to evaluate its accuracy at identifying regions derived through content-preserving modifications such as lossy compression, normalization, and dithering, while ignoring regions modified through content-altering effects such as pitch changes

and spliced-in audio from other sources.

The test cases used to evaluate audio analysis were derived from 5-minute clips of an excerpt from Vivaldi's *The Four Seasons*, a stand-up comedy show, an excerpt from the iconic "I Have A Dream" speech, and an undergraduate lecture. Starting with these four clips, we used the Sound eXchange (SoX) tool to generate derivations from the following transformations: extracting and removing subclips, splicing in other audio, inserting silences, applying lossy (MP3) compression, dithering, normalizing, and pitch altering.

For each test case, we examined the output of our analysis and compared it with ground truth to compute a proportion of actually-preserved regions that were correctly identified. We refer to these as *correct*. We also computed the proportion of modified regions that we incorrectly identified as preserved, which we label as *false*. The results are summarized in Table 2.2. A '-' is displayed in place of a correct percentage in those tests where no region of the audio clip was actually preserved from the original.

For variations of subclip splicing, our approach correctly identified all preserved regions of audio in all cases, demonstrating that it is as robust as naive matching on binary-encoded raw PCM samples. For dithering and normalization, our approach also achieved close to perfect results.

For MP3 compression, our analysis is fairly robust for those regions of audio where volume is not so high that many samples are clipped during transformation. In both the comedy show and the speech recordings, there were samples in which the volume was sufficiently high that clipping could not be avoided during compression and decompression. On these tests our success rate was lower, but our failure rate did not increase. However, for the music and the lecture clips which did not have such irregularities, our analysis proved very robust to compression.

We expect that a common use case for audio editing will be for an application

to both edit and compress a file, so we also evaluated our approach against a combination of compression and the other transformations. We created test cases by compressing the original audio clips to MP3 at 128 kbps, decompressing back to WAV, and then applying the other transformations. Analysis results are summarized in Table 2.3. As previously discussed, we are primarily concerned with keeping the false rate low. We note that for almost all tests, the false rate is close to zero while maintaining a high success rate.

Moreover, for the entire set of results, when we manually examined the regions which were falsely recognized as preserved, we observed that they were almost all moments of silence. We expect the data consumer to be able to recognize this when she listens to the actual derived audio.

2.7.2 Performance evaluation

Our primary goal in evaluating YouProve's performance was to measure the latency and power overheads of generating fidelity certificates. We expect the major contributor to be the computationally-expensive content analysis routines. We used our Nexus One YouProve prototype to produce fidelity certificates for a variety of data items, while measuring power using a Monsoon Power Monitor. Latencies are reported in Table 2.4. For photos ranging in size from 1296x972 to 2592x1944, it took just under 30 seconds to generate a fidelity certificate. For audio clips, latency ranged from about 20 seconds for a 30-second modified clip to about 64 seconds for a 5-minute clip. In all cases, the modified clip was encoded as MP3 and was compared against a 5-minute original clip. All reported data is an average over ten trials.

The average power consumption while generating fidelity certificates for both photo and audio content was relatively constant, between 1000 mW and 1100 mW for all trials. For comparison, we measured the power consumption of common tasks such as playing music (∼600 mW) and recording video (∼1800 mW). All measurements

Table 2.4: Latency of generating fidelity certificates.

Data item	**Latency in sec (stddev)**
JPEG, 1296x972	28.0 (0.12)
JPEG, 2592x1944	28.9 (0.34)
MP3, 30 sec	20.2 (0.12)
MP3, 60 sec	23.9 (0.59)
MP3, 5 min	64.1 (2.25)

were performed with the screen dimmed but not powered off.

The other question we sought to answer is whether any overheads imposed by YouProve negatively affect the user experience of using a mobile phone to gather sensor data. Specifically, does YouProve's logging increase the perceived latency of snapping a photo or recording an audio clip?

We found YouProve's impact on user experience to be minimal, presumably due to the asynchronous interfaces provided by Android's sensor APIs. Anecdotally, we did not perceive an increase to the delay of restoring the viewfinder after snapping a photo with the Camera app. While Android does not package a standard audio recording app, we did not notice any slowdown for the handful of recording apps we tested. The time to boot, or the delay from pressing the power button until the user interface becomes available, increased from 26 seconds with Nexus One stock firmware to 90 seconds for the YouProve prototype, mostly due to the SHA-1 digest computed over the read-only firmware.

We feel that these small and infrequent overheads are well worth YouProve's added authenticity guarantees.

2.8 Conclusions

This chapter has presented the design and implementation of YouProve, a system that enables mobile sensing services to verify that contributed data has not been manipulated in a way that alters its original meaning while allowing clients to use untrusted editing applications to directly control the fidelity of data they upload. Verifying that contributed data preserves the meaning of original sensor readings is a key requirement for ensuring data authenticity in domains such as citizen journalism. The key to YouProve's approach is providing analytic bases of trust for remote services. YouProve relies on trusted, content type-specific analyzers running on the mobile device to generate reports summarizing differences between a derived data item and an original sensor reading. Results of experiments with a prototype are promising. Logging source data does not noticeably affect application responsiveness, and our content analyzers are accurate and complete their tasks in under 70 seconds for 5-minute audio clips and under 30 seconds for 5-megapixel photos.

3

Dynamic Information Flow Tracking on Mobile Devices

This chapter describes TaintDroid, an extension to the Android operating system that enables dynamic information flow tracking, providing the capability to monitor how data is processed by untrusted code. TaintDroid enables precise tracking of information flows via system-wide dynamic taint analysis, or *taint tracking*. This capability can be utilized to monitor how third-party apps handle users' sensitive data, as well as to audit how sensor data is processed on the mobile device before being submitted to a mobile sensing service. The original TaintDroid system was a collaborative work of Enck, Gilbert, Chun, Cox, Jung, McDaniel, and Sheth [34]. This chapter begins with an overview of TaintDroid and then presents as contributions two key enhancements that improve the performance and precision of TaintDroid in Sections 3.2 and 3.3. These enhancements serve to enable follow-on work including a system for tracking implicit flows [29], a cloud-enabled system for large-scale analysis of mobile apps [38], and the approach for tracking modifications to image data described in Chapter 4.

3.1 TaintDroid

The massive popularity of smartphones and apps has led to an explosion of app development, with over 2 million apps available in the Apple's App Store [9] and 3 million apps in Google's Play Store [8] as of 2017. While apps can provide useful functionality and entertainment for mobile users, they also introduce unique privacy risks. They often access sensitive information from the phone's sensors like the GPS radio, camera, or microphone; or databases like the user's contacts list. While there are legitimate reasons for apps to access sensitive data, users are left with little awareness of how they actually use the data. There are many documented cases of apps unexpectedly uploading [53, 30] or otherwise mishandling sensitive data [12]. Unfortunately, mobile operating systems provide only coarse-grained controls for specifying the types of sensitive data an app is allowed to access. Granting access to a particular type of data is often required for an app to function properly. Once permission has been granted, the app may perform unwanted actions such as sending data back to its own servers or sharing it with third-party advertising or analytics services.

We address this problem with TaintDroid, an extension to the Android operating system that enables users to monitor how their sensitive data is used by third-party apps. TaintDroid implements *taint tracking* inside the Android OS to track flows of sensitive information in real time, from the source (typically an API that returns sensitive data), through the execution of an app, to the point where it is sent out over the network. The alerts generated when sensitive data is sent over the network provide users with valuable information that can help them make informed decisions about which apps to trust with their data.

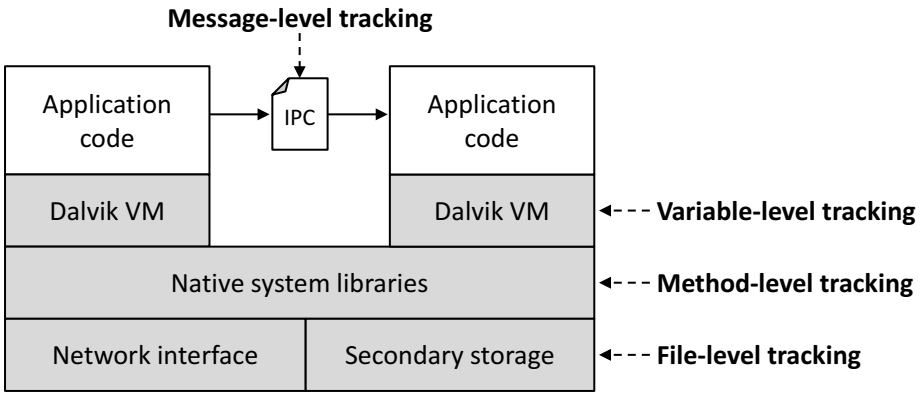

FIGURE 3.1: Taint tracking approaches for different Android components.

3.1.1 System overview

A key goal in the design of TaintDroid was that it should be able to track apps' use of sensitive data in real time, in order to support inspection and evaluation of apps by device owners or security analysts. As a result, TaintDroid must avoid the heavy performance costs incurred by previous taint tracking approaches that rely on full system emulation [27, 79]. We achieve this goal by implementing taint tracking at different granularities in different layers of the Android platform, tailoring our approach for the unique characteristics and performance demands of each component. While none of the techniques applied are novel, the combination of different techniques to form a comprehensive, practical solution for tracking information flows in a mobile OS is a key contribution.

Specifically, we implement *variable-level* taint tracking in Android's Dalvik VM for interpreted code. For native code, which is often used to implement performance-critical tasks, we avoid the performance penalties experienced by previous systems by providing *method-level* tracking. To track sensitive data when it leaves an app's address space, we provide *message-level* tracking for IPC messages and *file-level* tracking for sensitive data written to the filesystem. The relationships between various

Android components and their corresponding taint tracking approaches are shown in Figure 3.1.

Information flows are tracked from *taint sources*, which are typically APIs that return sensitive data, to a *taint sink*, invoked when sensitive data is passed to platform code used to send data over the network. When tainted data reaches the network taint sink, TaintDroid reports the source of the sensitive data, the contents of the network message, and the intended destination.

Taint tracking in the Dalvik VM

While Android apps are typically written in Java, Android's Dalvik VM executes Dalvik Executable (DEX) bytecode, a custom bytecode generated from Java bytecode at build-time using Android development tools. Dalvik uses a register-based architecture, unlike Java VMs, which are stack-based. A method in DEX defines a number of registers that correspond to local variables and are used to store primitive values and object references. All computation is performed on registers—values are read in from class fields, manipulated via DEX instructions, then written back to class fields.

TaintDroid tracks sensitive information flows in interpreted code by maintaining a *taint tag* for every register in a given Dalvik stack frame. The taint tag is a 32-bit value interpreted as a bit vector encoding up to 32 different *taint sources*. Examples of taint sources include the camera, microphone, and fine and coarse-grained location data. Taint tags are stored interleaved with register values in memory to maximize data locality. The modifications made by TaintDroid to the Dalvik stack format are depicted in Figure 3.2. Taint tag storage is shown in darker gray.

When the Dalvik VM performs an operation on one or more registers, the taint tags from the source registers are propagated to the taint tag for the destination register according to the semantics of the operation. For operations that take multiple

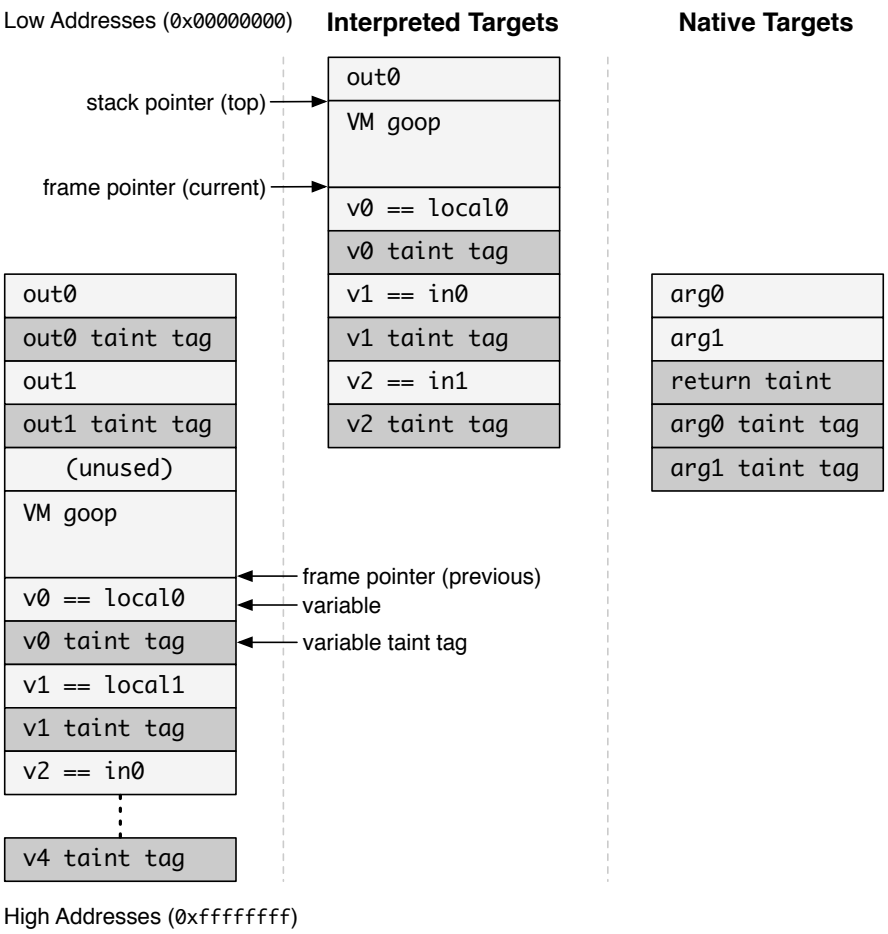

FIGURE 3.2: TaintDroid modified stack format.

source registers, this is typically as simple as propagating a bitwise OR of the source tags. Results of Dalvik operations are stored to class fields. To track information flows beyond the local method scope, we also allocate taint tags for all instance and static class fields.

This approach requires approximately 2x more storage for the Dalvik stack, and additional storage for objects proportional to their number of fields. Each DEX opcode handler must also perform additional computation to propagate taint tags from source to destination. The performance costs of the approach are discussed in Section 3.2.

Taint tracking for native code

A number of systems implementing taint tracking at the machine-instruction level have demonstrated that the performance costs of this approach are very high, with runtime overheads of 2x to 20x [79, 27]. As a result, we conclude that the approach is not compatible with our goal of real-time tracking. Instead, we provide tracking at the level of native methods rather than individual instructions. The performance overhead added by this approach is neglible, as it requires only a single taint propagation operation for each native method invocation—we simply propagate the taint tags of the method parameters to the return value and any parameter whose value has a data dependency on other parameters.

Method-level tracking represents a tradeoff, potentially sacrificing precision in return for real-time performance. In the case of native libraries provided by the Android platform, we avoid losing track of information flows by tailoring taint propagation for the data flow semantics of individual methods. For example, to enable tracking for the common routine `System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`, it is necessary to propagate the taint tag associated with the source array to the destination array. However, if we allow apps to load their own custom native libraries, we run the risk of failing to track information flows in that code that are not captured by the "parameter to return value" taint propagation heuristic. TaintDroid leaves the decision of whether to allow apps to load their own native libraries as a policy choice for the user.

3.2 Evaluating and improving TaintDroid performance

The main goals of the original TaintDroid work were to demonstrate that real-time taint tracking could be applied inside the Android OS to precisely track apps' use of sensitive data, and to validate the approach with a study of how popular apps

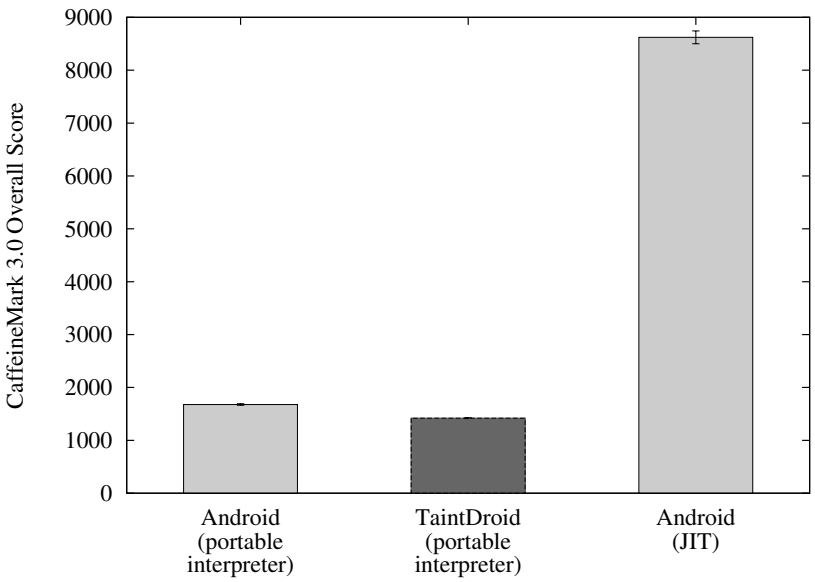

FIGURE 3.3: TaintDroid prototype Java performance compared to Android.

use different types of sensitive data. When building the first TaintDroid prototype, optimizing runtime performance was not a primary concern. As a result, taint propagation was implemented in the "portable" version of the Dalvik interpreter—a reference implementation, written in C, which was not intended to run on production devices.

Experiments comparing the performance of the TaintDroid prototype to the performance of the unmodified portable interpreter for running interpreted code suggested that the runtime overhead of taint propagation was only about 15%. However, because the portable interpreter is not optimized for performance, it was not clear whether this result was a meaningful representation of the overhead of taint tracking in a production mobile OS. Performance of the TaintDroid prototype is compared against that of Android's portable interpreter and the production version of the Dalvik VM, Android with JIT, in results shown in Figure 3.3. Scores are reported for CaffeineMark [16], a CPU-bound Java microbenchmark. The CaffeineMark score corresponds roughly to the number of operations that could be completed in a given

time and is only useful for relative comparisons. The results show that the TaintDroid prototype is approximately 6x slower than production Android when running interpreted code.

Because of the large gap in performance between the original TaintDroid prototype and production Android, we concluded that implementing taint tracking in the production version of the Dalvik VM would give a clearer picture of the performance costs of taint tracking in a consumer mobile OS.

3.2.1 Taint tracking in the production Dalvik VM

The production version of the bytecode interpreter used by Dalvik, referred to internally as the "fast" interpreter, achieves nearly 50% speedup over the portable interpreter, primarily through implementing each bytecode handler in hand-coded ARM assembly. With the release of Android 2.3, a just-in-time compiler (JIT) was added to the Dalvik VM to boost performance for bytecode execution [25]. JITs typically seek to improve performance by identifying frequently-executed bytecode sequences, and translating them to optimized native code at run-time. In the Dalvik VM, the interpreter profiles execution to identify candidate "hot" traces, which are subsequently compiled and optimized by the JIT. Within a trace, the JIT applies standard optimizations such as register promotion, redundant load/store elimination, and various loop optimizations. Once compiled and optimized, traces are stored in a per-process cache and re-used when the same execution path is traversed again. JITs typically provide the greatest performance gains for compute-intensive workloads—the Dalvik VM JIT achieves significant speedups of 2x to 5x for such workloads [25]. The introduction of the JIT was a major advancement for the Android platform, improving user experience and enabling developers to build more advanced functionality into their apps.

Enabling taint tracking in the production version of the Dalvik VM required im-

```
/* OP_ADD_INT vAA, vBB, vCC */
ldrh     r0, [rPC, 2]                @ r0<- CCBB
mov      r9, rINST, lsr #8           @ r9<- AA
mov      r3, r0, lsr #8              @ r3<- CC
and      r2, r0, #255                @ r2<- BB
ldr      r1, [rFP, r3, lsl #3]       @ r1<- vCC
ldr      r0, [rFP, r2, lsl #3]       @ r0<- vBB
add      r10, rFP, #4                @ r10<- rFP + 4
ldr      r3, [r10, r3, lsl #3]       @ r3<- tCC
ldr      r2, [r10, r2, lsl #3]       @ r3<- tBB
orr      r2, r3, r2                  @ r2<- tCC | tBB
str      r2, [r10, r9, lsl #3]       @ tAA<- r2
ldrh     rINST, [rPC, 4]!            @ advance rPC, load rINST
add      r0, r0, r1                  @ r0<- r0 + r1
and      ip, rINST, #255             @ extract opcode from rINST
str      r0, [rFP, r9, lsl #3]       @ vAA<- r0
add      pc, rIBASE, ip, lsl #6      @ jump to next instruction
```

FIGURE 3.4: Handler for `OP_ADD_INT` with taint propagation highlighted.

plementing taint propagation in the ARM assembly versions of bytecode handlers as well as the JIT. Figure 3.4 shows the bytecode handler for the `OP_ADD_INT` operation, which adds the values in the Dalvik registers denoted as B and C, and stores the result in register A. The extra assembly instructions needed to implement taint propagation are highlighted. A key reason for TaintDroid's relatively low runtime overhead compared to systems that perform taint tracking at the machine instruction level is apparent: performing taint propagation for a common operation like adding two integers requires only five additional instructions, including three memory operations that exhibit high data locality. The code for the handler increases in size from 11 instructions to 16 instructions. This is in contrast with machine-instruction level taint tracking, which typically requires several additional instructions to implement taint propagation for a single program instruction. This fact, along with the method-level approach for handling native code, is responsible for TaintDroid's

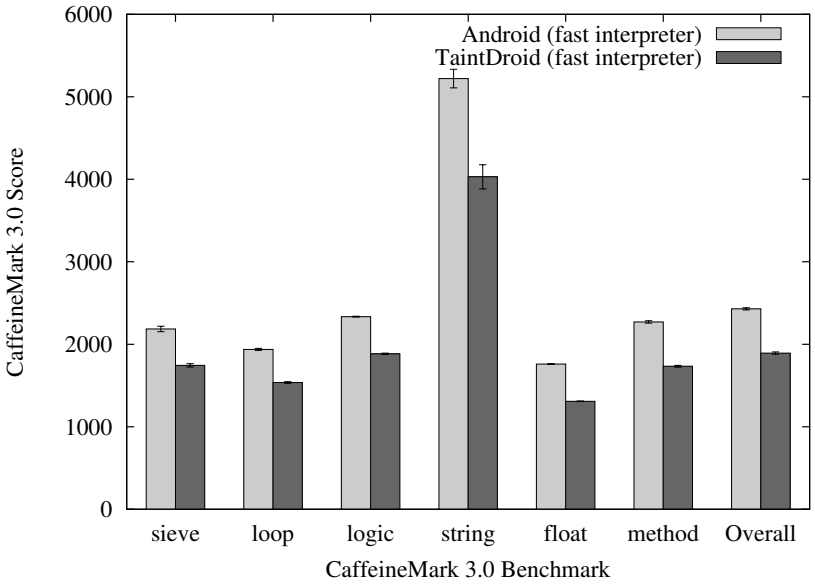

FIGURE 3.5: TaintDroid Java performance (fast interpreter).

speedy performance compared to other taint tracking systems.

Adding taint tracking to the JIT required modifying trace compilation—the process by which a sequence of DEX bytecodes is translated to an intermediate representation (IR) amenable to optimization. A handler for each bytecode instruction generates a sequence of IR instructions implementing the bytecode's logic. We modified these handlers to insert the same taint propagation logic implemented in the Dalvik interpreter. Once a bytecode sequence and its corresponding taint propagation logic have been translated to IR form, the JIT proceeds with the normal steps of optimizing the trace and compiling it to ARM native code.

3.2.2 TaintDroid performance results

Results for the CaffeineMark benchmark comparing TaintDroid using the fast Dalvik interpreter and the JIT with unmodified Android are shown in Figures 3.5 and 3.6, respectively. For all experiments, we used Android 4.1 and the corresponding version of TaintDroid running on a Samsung Galaxy Nexus phone.

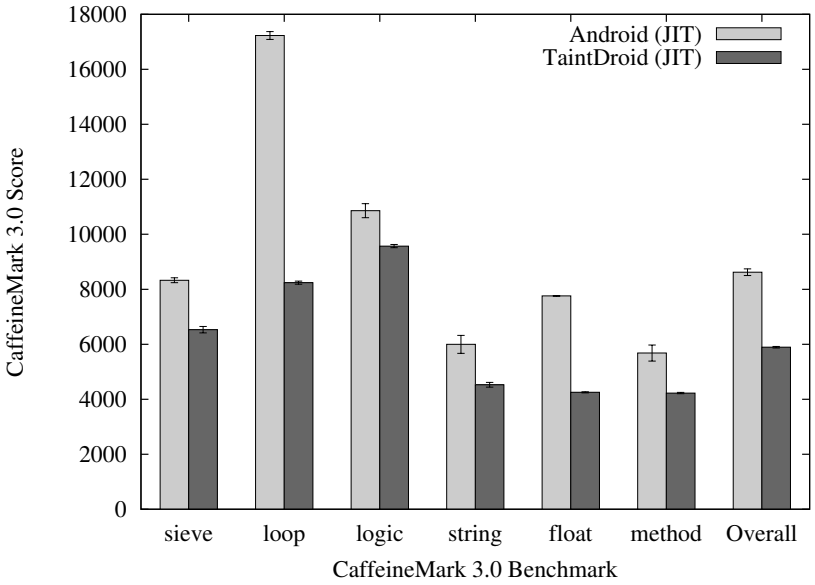

FIGURE 3.6: TaintDroid Java performance (JIT).

TaintDroid implemented in the fast interpreter is approximately 22% slower than the standard fast interpreter. The runtime overhead of taint propagation in the fast interpreter is somewhat higher than in the portable interpreter (15%). This result is not surprising, as the bytecode instruction handlers in the fast interpreter are optimized for performance, unlike those in the portable interpreter. In the fast interpreter version of TaintDroid, the additional instructions used to implement taint propagation make up a larger proportion of the total instructions needed to implement each bytecode handler.

The JIT version of TaintDroid is approximately 32% slower than the standard Android JIT. The performance impact of taint propagation in the JIT is greater than in the fast interpreter. This is likely due to unnecessary or unoptimized taint propagation code that remains after JIT optimizations eliminate or optimize other program logic. Taint tracking appears to be particularly costly for the `loop` benchmark, which experiences greater than 2x slowdown. It is not a coincidence that `loop` also experiences the greatest speed boost with JIT—in this case, the taint propaga-

tion code remaining after the substantial optimizations performed by the JIT has a disproportionate impact on TaintDroid's relative performance.

In the JIT implementation of TaintDroid, we did not make the various optimizations "taint-tracking aware." In other words, by the time a sequence of DEX bytecodes is converted to IR, the associated taint propagation logic for those bytecodes is decoupled and treated separately from the normal program logic. As a result, optimizations that improve performance for the DEX bytecode sequence are not automatically applied to the taint propagation code. It is possible that the performance cost of taint propagation in the JIT could be reduced by extending the JIT's performance optimizations to taint propagation logic. Such performance optimizations are a possible direction for future work.

3.3 Fine-grained tracking for arrays

Like TaintDroid's method-level approach for handling native code, its approach for tracking tainted data stored in arrays trades off a potential loss of precision for efficiency. TaintDroid stores a single taint tag per array to minimize storage overhead. Because byte and character arrays often account for a large percentage of an app's overall memory usage, allocating a 32-bit taint tag for every array element would result in a huge memory burden for apps running on already memory-constrained devices.

However, storing a single taint tag per array can lead to false positives, as any data read from an array containing both tainted and untainted data will be marked as tainted when returned by an `array-get` operation. Fortunately, experiments using TaintDroid to monitor how popular apps use sensitive data show that the approach does not lead to significant false positives from over-tainting. Manual validation was used to confirm that outgoing network traffic flagged by TaintDroid did indeed contain sensitive data [34].

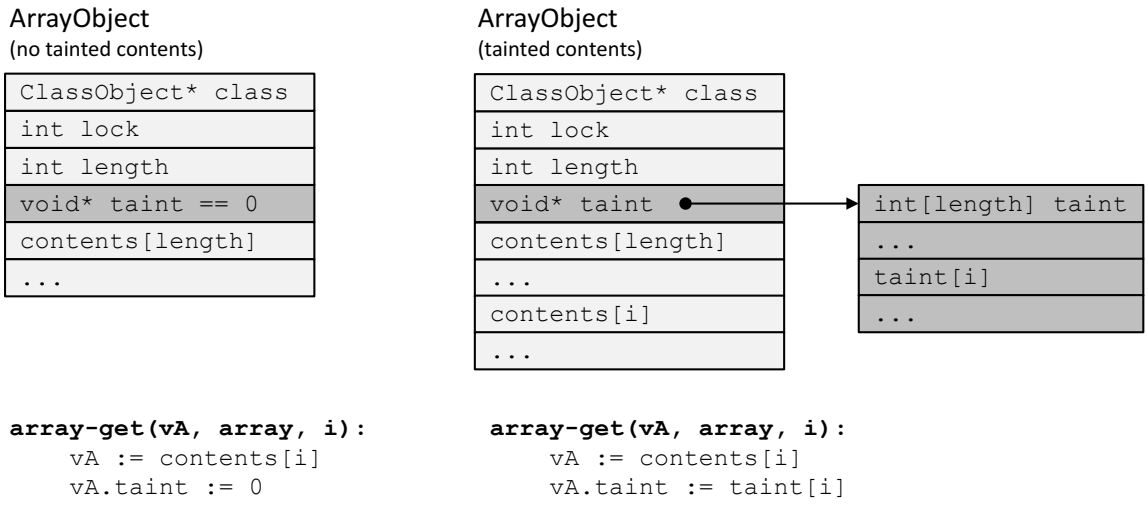

FIGURE 3.7: Internal structure of array object with fine-grained tracking.

While coarsed-grained tracking for arrays is appropriate for the initial use cases proposed for TaintDroid, per-element tracking for array contents is an important requirement for several systems that we have proposed building on the taint tracking platform provided by TaintDroid. In [29], we built an extension to TaintDroid to measure whether apps that handle password data leak information via implicit flows. In this work, it was crucial that we be able to track individual character values as they were moved among `Strings`, arrays, and local variables. Similarly, the approach for monitoring how apps modify image data presented in Chapter 4 requires tracking individual pixel values stored in arrays. For both of these systems, an approach was needed for tracking taint tags of individual array elements without imposing overly-burdensome memory overhead.

To balance these requirements, we implemented an alternative approach that allocates individual taint tags for all elements in a given array *on demand*, when tainted data is actually written to the array. Since the vast majority of arrays in typical programs are never used to store sensitive data, this approach imposes minimal storage or runtime overhead during normal execution. When operating on

untainted arrays, taint propagation is the same as in TaintDroid—a single null-valued taint tag indicates that no data in a given array is tainted. Additional overhead is incurred only when an app operates on arrays containing tainted data. Figure 3.7 shows the internal structure of an array object in the Dalvik VM before and after it is used to store tainted data. The logic for an `array-get` operation is shown for both cases.

3.4 Discussion

This chapter described the dynamic taint analysis system TaintDroid and presented two key enhancements that serve to enable several follow-on works. These enhancements significantly improve both the efficiency and precision of TaintDroid.

In the years since we built TaintDroid, an important change has been made to Android's approach for executing DEX bytecode. Starting with Android 5.0, the Dalvik VM was replaced with the Android Runtime (ART) [2]. Unlike Dalvik's interpreter-based approach, ART uses ahead-of-time (AOT) compilation to compile the DEX bytecode shipped by an app to machine code at the time of installation. ART delivers speedups of 1.5x to 2x and achieves both lower memory usage and longer battery lifetimes [23]. The costs of the AOT compilation approach are increased installation times and slightly more storage needed for the compiled application code.

To extend taint tracking to ART, a different approach is needed for propagating taint tags in application code. While it is no longer possible to insert taint propagation logic directly into the bytecode handlers used by the interpreter, we can take advantage of the fact that apps are still distributed as DEX bytecode and compiled on the device at install time. A number of systems have proposed enabling taint tracking by instrumenting Java bytecode [24, 21]. Similarly, we could rewrite an app's DEX bytecode to add taint propagation logic. Alternatively, it should be possible to achieve similar performance effiency to TaintDroid by adding taint propagation code

during the process of compiling DEX bytecodes to ARM instructions. Implementing and validating such an approach is a possible direction for future work.

4

Pixel Tracking for Characterizing Photo Modifications

This chapter describes an alternative approach for monitoring how image data is modified by apps through tracking operations performed on individual pixels. In contrast to YouProve, which as described in Chapter 2 uses an end-to-end approach to identify regions of content in a derived image that match regions in the original source data, the pixel tracking approach allows us to localize suspicious edits and characterize modifications by monitoring image processing code as it runs.

The system uses a modified version of the taint tracking platform TaintDroid to track modifications to individual pixels. The ability to track tainted data in arrays at per-element granularity rather than the coarse per-array granularity used in the original TaintDroid work, a contribution described in Chapter 3, is a key enabling feature.

4.1 Introduction

In Chapter 2, we presented YouProve and demonstrated that its photo analyzer is effective at identifying regions in an edited photo whose content is preserved from an

original photo. While this information is generally useful for assessing the authenticity of a photo, there are several other types of questions that we might want to answer about the photo's contents: For regions flagged by YouProve as not matching the original content, what types of manipulations were performed? How should we evaluate cases where a region's similarity score is close to the predetermined threshold? Was any external content combined with the original source photo to produce the final image?

We also observe that in some relatively common usage scenarios YouProve's photo analyzer can either fail to verify certain meaning-preserving transformations, or fail to provide enough information to determine whether a meaning-altering modification has been performed along with a benign transformation like scaling. For example, adjustments to brightness and contrast are difficult for YouProve's photo analyzer to handle. While minor adjustments typically do not compromise the meaning of the original content, they do change the values of most or all pixels. As a result, YouProve's photo analyzer tends to report high per-pixel SSD, or PPSSD, values for even minor adjustments, making it impossible to authenticate the final image.

YouProve's photo analyzer can also fail to deliver a conclusive verdict when images are scaled down in size. As the degree of downscaling is increased, the photo analyzer tends to produce increasing PPSSD values for some regions. This issue is discussed in Section 2.7.1. An attacker could take advantage of this phenomenon to game the system and fool a verifier consuming a fidelity report. The attacker could scale an original image down to the maximum degree such that some regions' PPSSD values approach but do not exceed the verifier's threshold. They could then introduce meaning-altering modifications to fine detail in regions with lower PPSSD values. The resulting fidelity report would not provide enough information for a verifier to distinguish between the regions affected by downscaling and those that were manipulated.

To address these cases where reports from YouProve's photo analyzer are not sufficient for determining how a photo was modified, we propose an alternative approach that monitors apps as they operate on photo data and tracks modifications to individual pixels. This approach, which we refer to as *pixel tracking*, enables us to localize suspicious edits to individual pixels in the final image and to characterize the types of modifications performed.

4.2 Pixel tracking

Pixel tracking applies taint tracking to maintain, along with each pixel value stored by an app, a *history* of the modifications performed on that pixel. We view this per-pixel history information as complementary to the end-to-end content similarity information provided by YouProve's photo analyzer. Pixel tracking results can be included as an additional analysis source in fidelity certificates, described in Section 2.4.5. In this secion, we describe how pixel history data is stored and tracked during the execution of image processing code.

4.2.1 Storing pixel histories

The natural structure for storing image data in memory to be displayed or edited is as an array of values representing the colors of individual pixels. A typical workflow in an Android photo editing app is first to obtain an `android.graphics.Bitmap` object from a compressed image file using an API like `BitmapFactory.decodeFile()`, to populate an array of pixel data via `Bitmap.getPixels()`, to perform modifications to that data in response to user inputs, and finally to apply the modifications using a variant of `Bitmap.setPixels()`.

During the entire lifetime of image data represented as pixel values inside an app, we store, along with each pixel value, a taint label that describes the history of that particular pixel. Unlike in TaintDroid, where a 32-bit taint tag represents a bit vector

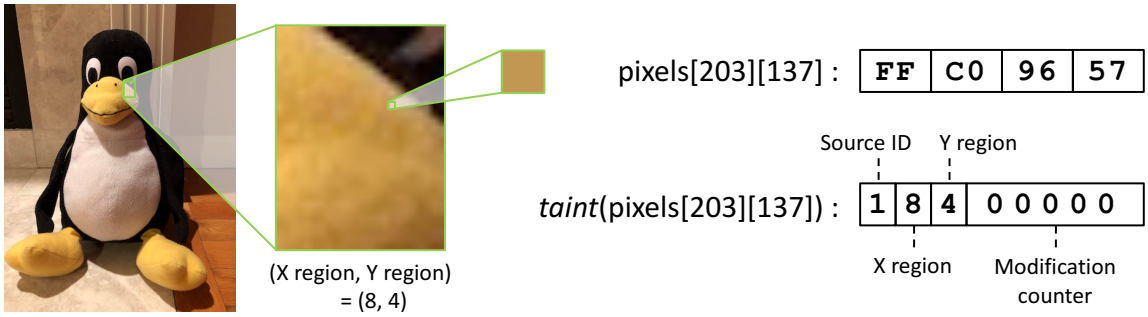

FIGURE 4.1: Pixel and corresponding taint label.

encoding up to 32 different taint sources, our taint label encodes (1) an ID that maps to the source image from which the pixel originated, (2) the pixel's original location on a 16x16 grid, and (3) a counter representing the number of times the pixel's value has been changed. A special ID value is used to flag the fact that the pixel's value was derived from data from two or more different sources images—this is critical for detecting attacks where external content is "pasted" in from a different image. An unmodified pixel and its corresponding taint label are shown in Figure 4.1.

4.2.2 Tracking pixel histories

Because the taint label associated with a pixel value encodes information about that pixel's history rather than a collection of taint sources as in TaintDroid, different logic is needed for propagating taint labels from the source(s) to the result of a given operation. Specifically, for any bytecode instruction that produces a result derived from one or more sources marked as a pixel value, we must propagate the source ID and region metadata as well as an updated counter value from the source pixel to the result. The logic for propagating pixel taint labels for a binary operator is depicted in pseudo-code in Figure 4.2.

Whenever an operation modifies a pixel, the modification counter associated with the resulting pixel is incremented. An important case that must be handled sepa-

```
t1 ← taint(src1)
t2 ← taint(src2)
if t1 == ∅ and t2 == ∅:
   // common case, untainted data
   taint(dest) ← ∅
else if t2 == ∅:
   // pixel data in src1 only
   taint(dest) ← incrementCounter(t1)
else if t1 == ∅:
   // pixel data in src2 only
   taint(dest) ← incrementCounter(t2)
else:
   // pixel data in both src1 and src2
   if id(t1) == id(t2):
     // src1 and src2 from same source image
     if counter(t1) > counter(t2):
       taint(dest) ← incrementCounter(t1)
     else:
       taint(dest) ← incrementCounter(t2)
   else:
     // src1 and src2 from different source images
     taint(dest) ← FLAG_MULTIPLE_SOURCE_IMAGES
```

FIGURE 4.2: Taint propagation logic for operation `dest ← src1 OP src2`.

rately is when an operation combines two or more pixels originating from different source images (i.e., labels with different source IDs). In this case, the ID for the resulting pixel is permanently set to a special value, indicating that the pixel was derived from multiple distinct source images—this is critical for detecting attacks where external content is pasted in from a different image.

4.2.3 Persistent storage

When written to persistent storage, image data is almost always transformed to a compressed format such as JPEG or PNG. Whether transformed into the frequency domain via JPEG encoding, or represented as a raster format like PNG, image data stored in a compressed format is not represented as an array of pixel values as during

processing. As a result, a separate approach is needed to preserve the information stored in pixel taint labels when image data is saved in compressed format.

One option for tracking across compression is to simply continue propagating pixel taint labels during the execution of the compression code, from the pixel data to the corresponding bytes in the compressed output. However, in experiments performed with JPEG compression, we found that the precise accounting of pixel modifications and their locations captured by pixel taint labels was lost during the transformation to the frequency domain and back. Although PNG compression does not transform image data across the frequency and spatial domains, the same issue exists of a lack of one-to-one correspondence between a particular pixel and bytes in the compressed representation.

To avoid losing the precision afforded by per-pixel taint labels, we instead add instrumentation to trusted platform library routines used to write `Bitmap` objects to files in compressed format (i.e., `Bitmap.compress()`), to capture a snapshot of the taint labels corresponding to the image's pixels. We save the taint label data, which is exactly the same size as the image's uncompressed form, and we generate a record binding the snapshot to the compressed image. To reduce the amount of extra space needed to store the snapshot, the snapshot could be compressed using a lossless compression scheme.

4.2.4 Image processing using the GPU

Apps often leverage the GPU for accelerating image processing code. "Filter"-style operations that perform the same local modification over wide regions of an image are particularly amenable to optimization via the GPU, as they are typically straightforward to parallelize. Unfortunately, our pixel tracking approach cannot track modifications to individual pixels made by code executing on the GPU. However, extending fine-grained tracking to GPU programs, commonly referred to as

shaders, may be possible by performing static analysis and/or instrumenting the code when an app requests it to be compiled and loaded onto the GPU. However, fine-grained tracking for modifications performed by GPU code is beyond the scope of this work.

We require that apps perform modifications in interpreted code to support fine-grained pixel tracking. However, we do provide the capability to make coarse-grained claims about image data modified by GPU code. Specifically, we instrument the OpenGL libraries used by third-party apps to interact with the GPU to record the source of any image data that might be present in an output image produced by the GPU. For example, an app might perform the following workflow to run a filter operation on the GPU: (1) obtain an OpenGL context, (2) compile and attach the shader implementing the filter logic, (3) load in the source image data as a *texture*, (4) execute the shader, and (5) copy the modified image data back to a `Bitmap` object. In this case, we can report the identity of any source images that were loaded into the OpenGL context as textures before the output image was generated. This allows us to cover the important case of content pasted in from another source image.

While the claims that our tracking system can make about image data modified using the GPU are not as detailed or precise as pixel tracking results, we believe that it is nonetheless useful to be able to verify that a final image contains pixel data from a single source image.

4.2.5 Handling native code

Apps often implement particularly heavyweight or performance-critical computations in native code, affording the developer more control over the way the computation is performed. We would like to extend pixel tracking to cover modifications performed using native code. However, to avoid performance overheads that could prohibit real-time tracking, TaintDroid provides only coarse-grained method-level tracking for

native functions. This approach provides no visibility into the operations performed on tainted data by third-party native code.

A number of recent systems have proposed extending fine-grained taint tracking to Android native code. DroidScope [78] and NDroid [61] both use an emulated Android environment based on the full-system emulator QEMU to monitor informations flows in native code. However, because pixel tracking is intended to run on real devices, we turn to SandTrap [64], an extension to TaintDroid that leverages features of the ARM processor architecture to enable instruction-level taint tracking for native code. SandTrap avoids the heavy performance penalties associated with emulating machine instructions by enabling emulation for individual threads on-demand, only when they actually access sensitive data. While this approach avoids slowdowns when native code does not touch sensitive data, performance penalties are inevitable when emulation is required. When native code is used to process image data, SandTrap incurs slowdowns of 6x to 8x [64]. As a result, while we can support pixel tracking in native code using SandTrap, we target apps that perform image processing in interpreted code, due to the high performance costs of emulating native image processing code.

4.2.6 Requirements for image editing apps

A key goal that influenced the design of YouProve was that it should support all third-party editing apps. However, in order to achieve precise per-pixel tracking and avoid unnecessary performance overheads, this approach imposes a few restrictions on image processing code. Most importantly, to recognize the full benefits of pixel tracking, apps must perform any modifications to image data in interpreted code. If an app uses the GPU to execute image processing code, we can perform only coarse-grained tracking, reporting only whether or not the output was potentially "contaminated" with data from another image. While we are able to track process-

ing performed by native code, any such processing will incur significant slowdown, invalidating the reason for using native code in the first place. The other requirement for image processing apps is that they use trusted platform APIs, i.e., our instrumented versions of `android.graphics.Bitmap` and `BitmapFactory`, to read and write image data to disk. This allows us to maintain pixel tracking data for image files stored in compressed formats.

These restrictions limit the options available to developers for optimizing the performance of their image editing code. However, we believe that the restrictions do not limit the range editing capabilities that apps can offer their users.

4.3 Evaluation

In evaluating pixel tracking, we sought to answer two questions: (1) Can pixel tracking aid in identifying the types of modifications performed on image data, including precisely locating meaning-altering local modifications? and (2) What are the performance costs of performing pixel tracking inside Android's Dalvik VM?

4.3.1 Identifying image modifications

To evaluate whether the information generated through pixel tracking is useful for determining the types of modifications performed on image data, we created a test app that implements a number of operations commonly offered by image editing apps. Operations included "global" filter-style effects like **blur**, **sharpen**, and **brightness/contrast** adjustment, as well as operations targeting specific regions, such as **pixelating** a subregion of the image and using a **paintbrush** tool. **Downscaling**, a particularly problematic operation for YouProve to handle, was also included. The implementations were taken directly from pseudo-code or actual example Java code found online, without any special accomodations to support pixel tracking.

In the figures showing pixel tracking results, a "heatmap"-style representation of

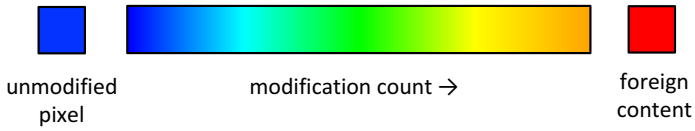

FIGURE 4.3: Key for pixel tracking heatmaps.

the pixel tracking results is shown to the right of each modified image. As noted in the key in Figure 4.3, blue, i.e., `(R,G,B) = (0,0,1)`, represents pixels reported as unmodified. Red represents pixels reported as not derived from the original source image—this could include solid colors, in the case of a paintbrush tool, or external data in the case of content pasted in from another image. The range of colors from blue to orange represent increasing values for modification count.

Following is a discussion of the pixel tracking results for each of the operations we considered and how the results could be used to characterize the type of modification performed.

Blur

Our blur operation is a standard Gaussian blur, which convolves the image with a Gaussian kernel. For efficiency, Gaussian blur is typically implemented in two passes, one each in the horizontal and vertical directions, rather than a single pass with a two-dimensional kernel.

The pixel tracking results for blur are shown in Figure 4.4. The data exhibits a gradient pattern that corresponds to the two-pass implementation. We observe smoothly increasing modification count values as we move along both the horizontal and vertical axes. This pattern is due to modification counts "accumulating" as pixels are modified in order from left to right and top to bottom during the two passes. The results suggest that smooth gradients observed in both the horizontal and vertical directions provide a useful "fingerprint" for identifying uniform two-pass global filters such as Gaussian blur.

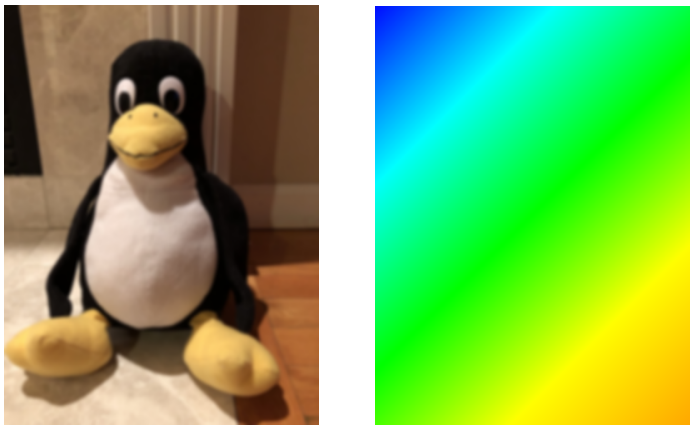

FIGURE 4.4: Pixel tracking results for Gaussian blur.

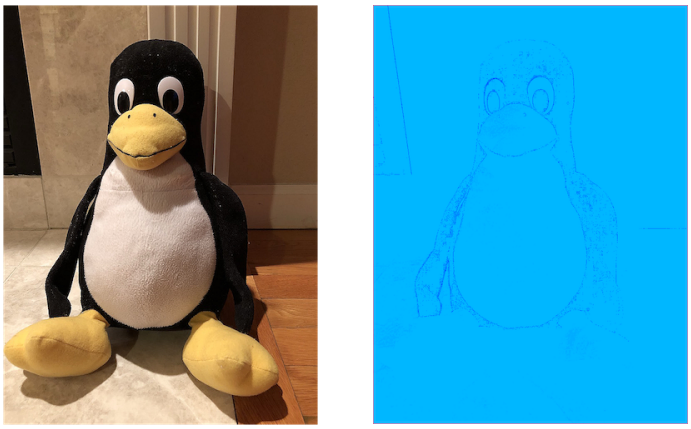

FIGURE 4.5: Pixel tracking results for sharpen.

Sharpen

Our sharpen operation computes a convolution of the original image with a 3x3 kernel. The pixel tracking results for sharpen are shown in Figure 4.5. The results show uniform modification counts across the entire image, with the exception of clusters of pixels with lower modification counts in areas where regions of black pixels border lighter colors. This phenomenon illustrates a side effect of tracking only explicit information flows. Due to the way the sharpen operation is implemented, some modified pixels exhibit a control-flow dependency on source data rather than

a data dependency. A simplified version of code that produces the final pixel values is as follows:

```
// sumR, sumG, sumB from matrix product of image data and kernel
// alpha from original image data
int R = (int)(sumR / sharpenFactor);
if (R < 0) { R = 0; }
else if (R > 255) { R = 255; }

int G = (int)(sumG / sharpenFactor);
if (G < 0) { G = 0; }
else if (G > 255) { G = 255; }

int B = (int)(sumB / sharpenFactor);
if (B < 0) { B = 0; }
else if (B > 255) { B = 255; }

pixel[x][y] = (alpha ≪ 24) | (R ≪ 16) | (G ≪ 8) | B;
```

In the code snippet, RGB values are clamped in the range [0, 255]. In cases where the values of sumR, sumG, and sumB are all less than zero, the final value assigned to the pixel has no data dependency on the result of multiplying the original pixels and the sharpen kernel. This case highlights the need to handle implicit flows in order to get a complete picture of how pixel data is modified.

A number of possible approaches exist for tracking implicit flows. A simple solution would be to disallow image editing code from branching on conditions involving pixel values, except in special white-listed functions for common operations like clamping a value to a specified range. This approach would be suitable for cases like this sharpen operation. We propose a less-restrictive approach for tracking implicit flows in [29].

Brightness/contrast adjustment

In this experiment, we used our brightness/contrast operation to increase both brightness and contrast to a moderate degree. Pixel tracking results are shown

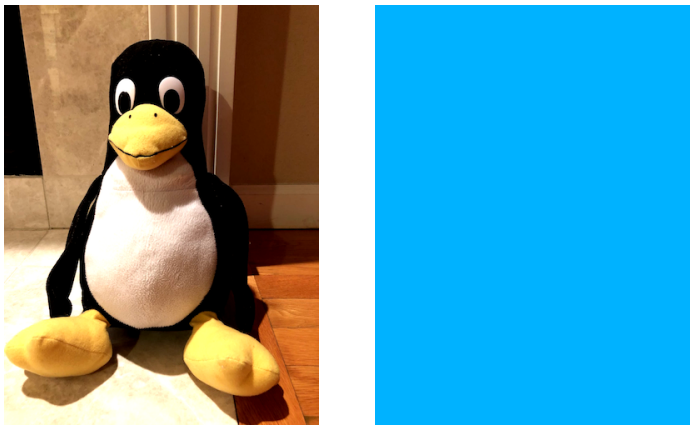

FIGURE 4.6: Pixel tracking results for brightness/contrast adjustment.

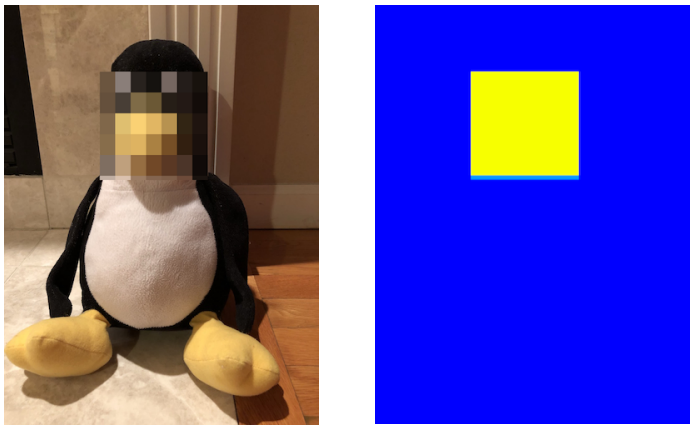

FIGURE 4.7: Pixel tracking results for pixelate.

in Figure 4.6. As the exact same computation is performed on every pixel, we observe completely uniform modification counts across the entire image. This result demonstrates that pixel tracking can be useful for characterizing a popular meaning-preserving modification that YouProve's photo analyzer cannot handle.

Pixelate

Results for pixelating a region of an image are shown in Figure 4.7. We observe uniform modification counts throughout the targeted region, while the rest of the image is correctly reported as being unmodified. The horizontal band of pixels across

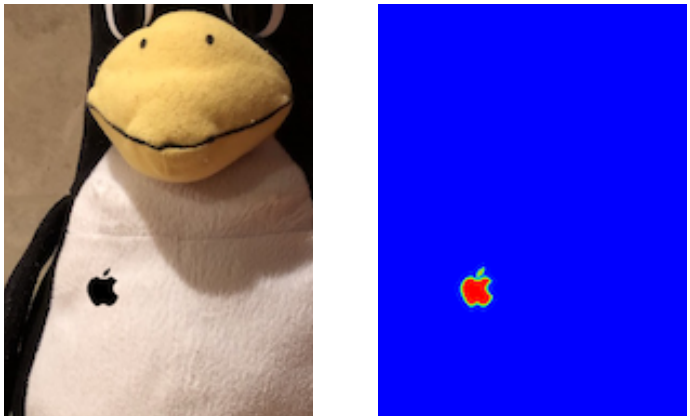

FIGURE 4.8: Pixel tracking results for paintbrush operation.

the bottom of the modified region with lower modification counts is a result of the pixelation algorithm not dividing the region into equally-sized blocks. The color of each block is determined by averaging the values of its pixels in the original image. The smaller-sized blocks at the bottom of the modified region contain fewer pixels, therefore requiring fewer operations to compute the average. This result suggests that operations that apply a uniform transformation across a region of an image can be identified by regions of uniform modification counts in the pixel tracking data.

Paintbrush

Results for the paintbrush operation, displayed in Figure 4.8, show that pixel tracking can precisely localize potentially meaning-altering modifications to the specific modified pixels. Our paintbrush operation is intended to emulate the use of a "feathered brush" or airbrush tool common in image editing software. Specifically, the central portion of the brush replaces the targeted pixels with a solid color, while the color is blended with the underlying image content at the edges. In our experiment, the shape of the stenciled apple is apparent in the pixel tracking data. The pixels that were painted solid black appear red in the heatmap, indicating content not originating from the source image. The pixels that were blended around the edges have

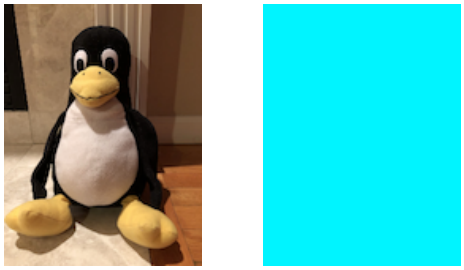

FIGURE 4.9: Pixel tracking results for downscale operation.

non-zero modification counts.

Downscale

Detecting possible meaning-altering modifications in the presence of downscaling was an important motivating case for pixel tracking. Many different algorithms exist for scaling image content—we chose bilinear interpolation for our downscale operation. The pixel tracking results for downscaling an image to 33% of its original width/height are shown in Figure 4.9. As with other operations that perform uniform processing over image data, the modification counts reported by pixel tracking are highly uniform across the entire image. This is in contrast to YouProve's photo analyzer, which often reports non-uniform elevated PPSSD values across various regions of a downscaled image. The pixel tracking results suggest that the approach will be useful for identifying other manipulations performed along with downscaling, as downscaling introduces no "noise" into the pixel tracking data.

4.3.2 Performance evaluation

To evaluate the performance costs of pixel tracking, we compared the performance of our test image editing application on a Dalvik VM with pixel tracking enabled, versus a VM with traditional taint tracking enabled (i.e., TaintDroid) and an unmodified Dalvik VM. We measured both the latency of executing various image processing operations as well as the memory usage of the app. All experiments were performed

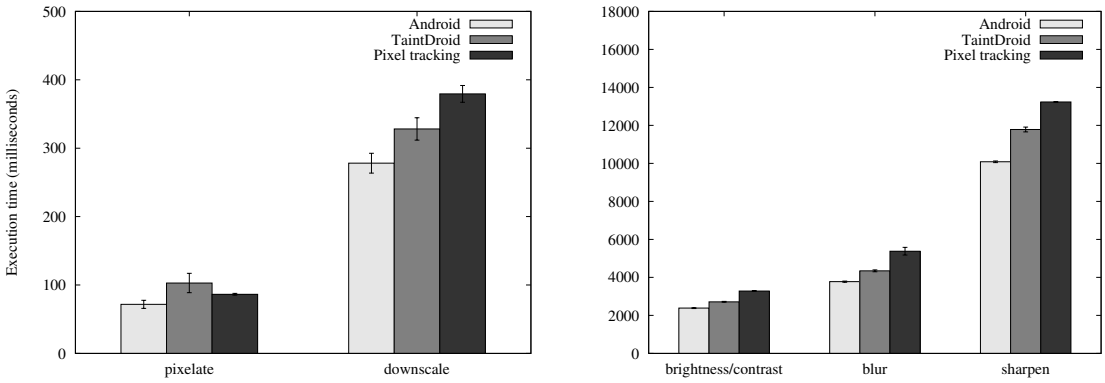

FIGURE 4.10: Execution time for image processing operations.

using the portable version of the Dalvik interpreter, as it is the version used in the pixel tracking prototype.

Results reporting the time needed to complete various image processing operations on a photo measuring 756x1008 pixels are shown in Figure 4.10. Operations are generally slower when using pixel tracking than with standard TaintDroid, with both taint tracking systems being slower than stock Android. The lone exception was that our measurements of the fastest operation, pixelate, did not confirm it to be faster under Taintdroid than with pixel tracking. This is likely due to variance added by outside factors such as garbage collection having a greater impact when measuring shorter time intervals.

Overall, pixel tracking made executing image processing operations approximately 21% to 43% slower than when no taint tracking was performed. This result suggests that we can obtain the benefits of the fine-grained per-pixel tracking without excessively slowing down image processing apps.

In addition to execution time, we also measured the additional memory usage of our test app when operating on image data. The "high water mark" of the app's memory usage when performing various operations is shown in Figure 4.11. As expected, slightly more memory was used by Taintdroid than by Android, as a

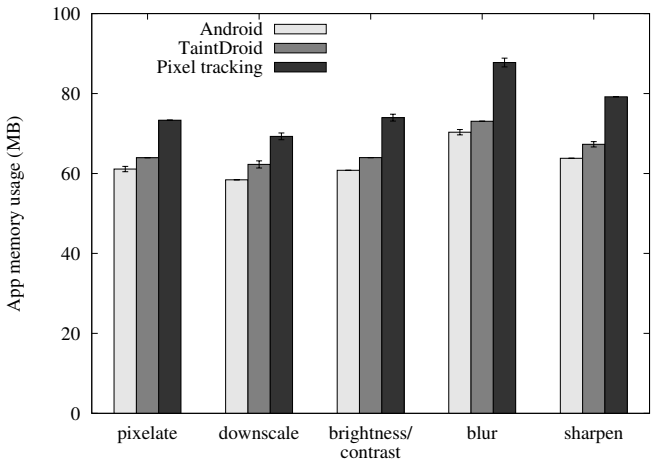

FIGURE 4.11: Memory used for image processing operations.

four-byte taint tag is allocated for every variable and field in the Dalvik VM. Pixel tracking used significantly more memory: between 20% and 25% more than Android. This is expected, as pixel tracking requires extra taint storage for every pixel. The amount of additional memory needed for taint storage will ultimately depend on how an image processing app chooses to manage image data in memory—extra copies of image data, which may be used to support "undo" functionality, require more taint storage.

4.4 Conclusions

In this chapter, we presented pixel tracking, an approach for monitoring how apps modify image data that complements the end-to-end similarity analysis performed by YouProve. We showed that pixel tracking can be used to precisely localize manipulations to image content, and that it can provide useful information in cases where YouProve's image analyzer falls short. Evaluation results also show that for common image processing operations, pixel tracking data can be useful for characterizing the type of operation performed. It is particularly useful for differentiating between uniform filter-style effects, that operate over entire regions independent of

the content, and local modifications that are likely to change how the content is perceived. Experiments with a prototype implementation in the Dalvik VM suggest that the runtime and memory overheads imposed by pixel tracking do not preclude its use for real-time monitoring.

5

Related Work

There is a vast amount of work related to this dissertation in a number of different areas. The proceeding discussion of related work is organized into the following categories: data fidelity in mobile systems, approaches for verifying data authenticity, systems that utilize trusted hardware, information flow tracking, and approaches for detecting manipulated photos.

5.1 Data fidelity in mobile systems

Fidelity has traditionally been studied in the context of mobile clients retrieving data from servers over a wireless network [36, 48, 59]. In these settings, a small set of trusted servers maintain canonical copies of all source data and can generate reduced-fidelity versions at the request of a client. In a mobile sensing service, the roles of clients and servers are reversed: clients use sensors to generate original data and may reduce its fidelity locally before sending it to a server which operates on the data to implement the service logic.

YouProve's type-specific analyzers are reminiscent of Odyssey's type-specific wardens [59]. Each data item under Odyssey's control is assigned a type (e.g., video,

audio, or photo) and is stored by the server at multiple fidelity levels. An Odyssey warden is a client-side software component that interacts with a server to retrieve data of the appropriate fidelity. YouProve's data analyzers are similar to Odyssey's wardens in that they are also data-centric.

5.2 Verifying data authenticity

Several groups have identified data authenticity as a critical problem in mobile sensing. Dua et al. [33] proposed pairing a mobile device with a trustworthy sensing peripheral that can attest to its software configuration and sign its sensor readings. This approach only applies to raw sensor readings and cannot ensure the authenticity of locally modified data. Other groups have proposed deploying trustworthy signing infrastructure that can accept sensor readings from nearby devices and provide signed timestamps and location coordinates [50, 66]. However, this approach can only prove that a data item existed at a particular time and place.

Two position papers made the case for trustworthy sensing on mobile devices [39, 67]. Saroiu et al. [67] describe two architectures for signing sensor readings from a device. One is similar to Dua's approach [33] and embeds signing hardware in the sensors themselves so that services do not need to make any trust assumptions about a device's software. The other proposed approach utilizes a TPM and virtual machines (VMs) to minimize the TCB by including sensor drivers in the hypervisor and placing all other functionality in untrusted VMs. Not-a-bot [41] used a similar architecture to minimize the TCB and certify when outgoing network messages were temporally correlated with keyboard activity. Both approaches provide a smaller attack surface than YouProve, but neither provides authenticity guarantees for modified data, which is our primary goal.

We previously proposed a different architecture that utilizes VMs and a TPM [39]. In this approach an application that is trusted to modify sensor data is encapsulated

within a VM along with any relevant device drivers. This approach is similar to Saroiu's [67] virtual-machine proposal, but increases the size of the TCB to include code for modifying sensor data. As we discussed in Section 2.2.3, the main disadvantage of this approach is that it limits choice by forcing users to modify their sensor data with applications that a service deems trustworthy.

5.3 Utilizing trusted hardware

There have been several recent projects aimed at designing new trustworthy software architectures leveraging a TPM [52, 56, 68]. Nauman et al. [56] describe how a TPM can be leveraged to attest to the individual Java classes that Android's Dalvik VM loads. Nexus [68] is a micro-kernel OS for TPM-enabled machines that provides a general framework for generating and reasoning about certifiable statements. YouProve could be implemented on top of Nexus. Flicker [52] can attest to having executed a program, including its inputs and outputs, even if the machine's BIOS and OS have become compromised. It does this by relying on a small TCB of hundreds of lines of code and the late launch feature of a TPM. We could leverage architectures such as Flicker to provide stronger integrity guarantees for YouProve's sensor log and fidelity certificates.

CertiPics [68] is a trusted image editor for Nexus structured as a pipeline of small, stand-alone programs implementing a single operation such as cropping or resizing. To capture how an image was modified, each program invokes the Nexus kernel to generate a signed statement describing the hash of the program and a semantically-rich description of the operation it performed. As mentioned in Section 2.2.3, this approach either limits users' choice or places an unmanageable trust-management burden on services.

The Trusted Language Runtime (TLR) [65] provides a way to partition a mobile-phone application between a software component called a trustlet, which requires

access to sensitive data, and the rest of the app. The TLR runs within the TrustZone of an ARM processor, and applications can request that the TLR execute their trustlet within an isolated sandbox within the TrustZone called a trustbox. TLR provides an appealing programming abstraction for developers of apps that handle sensitive information (i.e., banking or payment apps). However, it does not obviously provide a way to resolve the tension between data authenticity and fidelity that motivates our approaches.

5.4 Tracking information flows

Many systems have employed dynamic information flow tracking to enforce access control based on the Denning's seminal work on a lattice model for secure information flow [31]. Abestos [73], Flume [47], and HiStar [80] all track information flows at the OS level, at the granularity of processes and OS-managed resources such as files and sockets. Process-level tracking is not precise enough to provide useful feedback about how mobile apps handle sensitive data.

Dynamic taint analysis, or taint tracking, has been used extensively for tracking information flows in legacy code. Dytan [28], TaintCheck [57], and LIFT [62] use taint tracking to defend against attacks on system security. Like TaintDroid, Panorama [79] and Privacy Scope [83] utilize taint tracking to detect sensitive data leaks. Taint tracking has been implemented at the system level using hardware extensions [71, 72] and emulation platforms [27, 79]. Other approaches have used dynamic binary translation to perform process-level tracking [28, 26, 62]. Taint tracking has also been added to VMs and interpreters [24, 42, 55, 74]. TaintDroid's approach for tracking information flows in the Dalvik VM is similar to some of these systems, but TaintDroid differs in that it extends taint tracking to other system components to provide OS-wide tracking.

McCamant and Ernst proposed a technique based on network flow capacity to

determine how much information about a program's sensitive inputs is revealed by its outputs [51]. They demonstrated their approach by quantifying the amount of information preserved by various image transformations such as pixelating, blurring, and twisting. This approach can quantify the total amount of information in bits preserved by these modifications, while pixel tracking attempts to characterize the modifications performed on specific pixels.

The concept of "positive tainting" introduced by WASP [43] is similar in purpose to pixel tracking. WASP uses positive tainting of characters to evaluate the integrity of SQL queries. The pixel histories maintained in pixel tracking could be thought of as positive taints for source photo data.

5.5 Detecting manipulated photos

Digital image forensics experts use a number of techniques to investigate photos suspected of being manipulated. These approaches are often subjective and non-scientific, e.g., manually examining whether shadows are consistent with light sources. These approaches can be useful, but ultimately they can be fooled by an informed attacker.

Error level analysis is a more systematic forensic approach proposed for detecting manipulations to images that have been compressed with lossy compression [46]. It involves looking for inconsistencies in compression artifacts across different parts of an image, and it often involves subjecting the image to further rounds of compression to make any differences more prominent. Forensic approaches like error level analysis, which are typically applied after the fact, without any information about the original content, are complementary to the analyses performed by YouProve's photo analyzer and pixel tracking. The results of error level analysis may help identify suspicious images, but they are still ultimately subjective and not sufficient to guarantee that an image has or has not been altered. In fact, claims based on error level analysis

have been a source of controversy in several high-profile cases [5, 3].

Fragile and semi-fragile watermarks [35] are types of digital watermarks proposed for detecting image manipulations. A more common type of digital watermark is a robust watermark, which is typically designed to be preserved across broad classes of modifications. Robust watermarks are used for intelluctual property and copy protection. Fragile watermarks are designed to indicate whether any modification has been performed—they are typically used for tamper detection and address a similar problem as message digests. Semi-fragile watermarks [35] are most closely related to our work, as they are designed to be resistant to content-preserving modifications and to support detection of malicious modifications. However, in general semi-fragile watermarks cannot provide guarantees in the presence of adversaries who are aware of the watermark structure. An informed attacker may be able to extract the original watermark, perform any desired modifications, and then reapply the watermark to evade detection. While several approaches have sought to improve the security properties of semi-fragile watermarks [82, 77], we are not aware of any approach that has demonstrated both strong security guarantees and accurate detection.

6

Conclusion

This dissertation makes several contributions that serve to validate its thesis:

- YouProve demonstrates that it is possible to provide guarantees about the authenticity of user-contributed data without restricting users' freedom to perform necessary modifications using any editing tools they choose.
- Pixel tracking provides useful information for characterizing modifications performed on photos, especially in important cases where YouProve cannot conclusively authenticate the data.
- Two key enhancements to the taint tracking system TaintDroid make it suitable for use in pixel tracking and other follow-on work.

Experiments with our prototype implementations demonstrate that all approaches are feasible and can achieve real-time performance on actual mobile phones.

Bibliography

[1] Antifa Member Photographed Beating Police Officer? https://www.snopes.com/antifa-member-photographed-beating-police-officer/.

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

[3] 'Bellingcat Report Doesn't Prove Anything': Expert Criticizes Allegations of Russian MH17 Manipulation. http://www.spiegel.de/international/world/expert-criticizes-allegations-of-russian-mh17-manipulation-a-1037125.html.

[4] Fake images from the Catalan referendum shared on social media. https://elpais.com/elpais/2017/10/06/inenglish/1507278297_702753.html.

[5] 'Fake' World Press Photo isn't fake, is lesson in need for forensic restraint. https://www.wired.co.uk/article/photo-faking-controversy.

[6] Kenyan app Ushahidi monitoring US elections. http://www.bbc.com/news/world-africa-37910068.

[7] No, the shark picture isn't real: A running list of Harvey's viral hoaxes. https://www.washingtonpost.com/amphtml/news/the-intersect/wp/2017/08/28/no-the-shark-picture-isnt-real-a-running-list-of-harveys-viral-hoaxes/.

[8] Number of available applications in the Google Play Store from December 2009 to September 2017. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[9] Number of available apps in the Apple App Store from July 2008 to January 2017. https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/.

[10] TPM Emulator. http://tpm-emulator.berlios.de/.

[11] TrouSerS. http://trousers.sourceforge.net/.

[12] WhatsApp user chats on Android liable to theft due to file system flaw. https://www.theguardian.com/technology/2014/mar/12/whatsapp-android-users-chats-theft.

[13] WhatsApp with FinSpy? http://maldr0id.blogspot.com/2014/10/whatsapp-with-finspy.html.

[14] When I share a photo on Instagram, what's the image resolution? https://help.instagram.com/1631821640426723.

[15] Network Time Protocol (Version 3) Specification, Implementation and Analysis. IETF, March 1992.

[16] CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/, 1997.

[17] ARM Security Technology: Building a Secure System using TrustZone Technology. ARM Technical Paper, 2009.

[18] Trusted Computing Group (TCG) Mobile Trusted Module Specification 1.0, version 7.02. TCG Published, April 2010.

[19] X. Bao, T. Narayan, A. A. Sani, W. Richter, R. R. Choudhury, L. Zhong, and M. Satyanarayanan. The Case for Context-Aware Compression. In *HotMobile*, 2011.

[20] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110:346–359, June 2008.

[21] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

[22] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Rmanathan, S. Reddy, and M. B. Srivastava. Participatory Sensing. In *World-Sensor-Web*, 2006.

[23] B. Carlstrom, A. Ghuloum, and I. Rogers. The ART runtime. In *Google I/O*, 2014.

[24] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 463–475, Dec 2007.

[25] B. Cheng and B. Buzbee. A JIT Compiler for Android's Dalvik VM. In *Google I/O*, 2010.

[26] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dyanmic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pages 749–754, June 2006.

[27] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[28] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[29] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. SpanDex: Secure Password Tracking for Android. In *USENIX Security*, 2014.

[30] C. Davies. iPhone spyware debated as app library "phones home". `http://www.slashgear.com/iphone-spyware-debated-as-app- library-phones-home-1752491/`, August 17, 2009.

[31] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[32] J. R. Douceur. The Sybil Attack. In *IPTPS*, 2001.

[33] A. Dua, N. Bulusu, W. chang Feng, and W. Hu. Towards Trustworthy Participatory Sensing. In *HotSec*, 2009.

[34] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[35] E. J. D. Eugene T. Lin, Christine I. Podilchuk. Detection of Image Alterations Using Semifragile Watermarks. 2000.

[36] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *ASPLOS*, 1996.

[37] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Micro-Blog: Sharing and Querying Content Through Mobile Phones and Social Participation. In *MobiSys*, 2008.

[38] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS '11, pages 21–26, New York, NY, USA, 2011. ACM.

[39] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward Trustworthy Mobile Sensing. In *HotMobile*, 2010.

[40] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *MobiSys*, 2003.

[41] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *NSDI*, 2009.

[42] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, December 2005.

[43] W. G. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[44] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual Trip Lines for Distributed Privacy-Preserving Traffic Monitoring. In *MobiSys*, 2008.

[45] J. Kincaid. Foursquare Starts To Enforce The Rules, Cracks Down On Fake Check-Ins. `https://techcrunch.com/2010/04/07/foursquare-starts-to-enforce-the-rules-cracks-down-on-fake-check-ins/`, April 2010.

[46] N. Krawetz. A Picture's Worth... Digital Image Analysis and Forensics. In *Presented at Black Hat Briefings USA*, 2007.

[47] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2007.

[48] E. D. Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *USITS*, 2001.

[49] W. Laube. Sickening tsunami of faked photos. The Sydney Morning Herald, March 2011.

[50] V. Lenders, E. Koukoumidis, P. Zhang, and M. Martonosi. Location-based Trust for Mobile User-generated Content: Applications, Challenges and Implementations. In *HotMobile*, 2008.

[51] S. McCamant and M. D. Ernst. Quantitative Information Flow As Network Flow Capacity. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 193–205, New York, NY, USA, 2008. ACM.

[52] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[53] D. Moren. Retrievable iPhone numbers mean potential privacy issues. `http://www.macworld.com/article/143047/2009/09/phone_hole.html`, September 29, 2009.

[54] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda. PEIR, the Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research. In *MobiSys*, 2009.

[55] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A Virtual Machine Based Information Flow Control System for Policy Enforcement. In *the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM)*, 2007.

[56] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In *TRUST*, 2010.

[57] J. Newsome and D. Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *NDSS*, 2005.

[58] E. M. Newton, L. Sweeney, and B. Malin. Preserving Privacy by De-Identifying Face Images. *IEEE Transactions on Knowledge and Data Engineering*, 17, February 2005.

[59] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *SOSP*, 1997.

[60] S. Odio. More Beautiful Photos. Official Facebook Blog, September 2010.

[61] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. On tracking information flows through jni in android applications. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 180–191, Washington, DC, USA, 2014. IEEE Computer Society.

[62] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[63] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *USENIX Security*, 2016.

[64] A. Razeen and L. P. Cox. Sandtrap technical report. 2017.

[65] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *HotMobile*, 2011.

[66] S. Saroiu and A. Wolman. Enabling New Mobile Applications with Location Proofs. In *HotMobile*, 2009.

[67] S. Saroiu and A. Wolman. I am a Sensor, and I Approve This Message. In *HotMobile*, 2010.

[68] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1), May 2011.

[69] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981.

[70] S. J. Snyder. Gotham Tornado: Amazing Photo of Twister Passing Statue of Liberty. Time Newsfeed, September 2010.

[71] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2004.

[72] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, 2004.

[73] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)*, 25(4), December 2007.

[74] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. of Network & Distributed System Security*, 2007.

[75] A. Wang. An Industrial Strength Audio Search Algorithm. In *ISMIR*, 2003.

[76] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *STC Workshop*, Oct. 2008.

[77] X. Wu, J. Hu, Z. Gu, and J. Huang. A Secure Semi-fragile Watermarking for Image Authentication Based on Integer Wavelet Transform with Parameters. In *Proceedings of the 2005 Australasian Workshop on Grid Computing and e-Research - Volume 44*, ACSW Frontiers '05, pages 75–80, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[78] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, 2012. USENIX.

[79] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM Computer and Communications Security*, 2007.

[80] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[81] X. Zhang, O. Aciiçmez, and J.-P. Seifert. Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms. In *MobiSec*, June 2009.

[82] X. Zhou, X. Duan, and D. Wang. A Semifragile Watermark Scheme for Image Authentication. In *10th International Multimedia Modelling Conference, 2004. Proceedings.*, pages 374–377, Jan 2004.

[83] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Technical Report EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.

Biography

Peter J. Gilbert was born in Durham, North Carolina on June 27, 1984. He defended his PhD dissertation at Duke University in December 2017. He received a B.S. in Computer Science from Mississippi State University in 2006. His research interests include mobile computing, and security and privacy.