DATA PROTECTION VIA VIRTUAL MICRO SECURITY PERIMETERS

By

GABRIEL SALLES-LOUSTAU

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Saman Zonouz

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

MAY, 2018

ABSTRACT OF THE DISSERTATION

Data Protection via Virtual Micro Security Perimeters

by Gabriel Salles-Loustau

Dissertation Director:

Saman Zonouz

Mobile devices have become the platform of reference for data consumption. Between personal and work related usages, users entrust their mobile devices to handle data from different sources with different sensitivity. Unfortunately, mobile device platforms are not designed to accommodate these usages and fail to provide adequate security mechanisms to guaranty users data protection or even isolation across sources.

This thesis focuses on client-oriented data protection solutions for embedded devices and more specifically smartphone-based operating systems. Three main aspects are explored.

First, this thesis introduces the concept of virtual micro security perimeters, or in short data *capsules*, as a new primitive to track and protect user data on smartphone devices. Data capsules consist in a set of data associated to a specific provenance or to a specific device usage (e.g., work vs personal). Contrary to security through compartmentalization solutions that often provide an inflexible isolation for data or execution environments, capsules leverage information flow tracking techniques as a primitive to track and protect capsules data. This approach enables the use of any application the user might like to access data of different sensitivity while still providing

strong data protection guaranties. We present an implementation and an evaluation of this approach through a prototype developed on top of the Android operating system.

Second, we propose a new approach to detect sensor-based data flows via the inspection of numerical operations and their operands. This approach uses numerical operations computed values as a flow detection mechanism rather than labels or taints that are commonly used in information flow tracking systems. We evaluate our approach through the implementation of a prototype that run as a third-party application and that does not require any system changes. This solution generates a minimal computation and space overhead while not sacrificing the flow detection accuracy.

Finally, we present a data protection solution for point-of-care devices that greatly reduce the trusted computing-based for data protection by using a hardware-based domain specific scrambling mechanism for point-of-care medical devices.

Acknowledgements

I would like first to thank my doctoral committee: Professors Dario Pompili, Marco Gruteser, Janne Lindqvist, and Doctor Kaustubh Joshi for their valuable feedback.

I would like to thank Robin Berthier, without whom this amazing journey would have not been possible, as well as my former intern advisors at University of Maryland: Michel Cukier and Danielle Chrun. I would also like to thank my former professors at the ENSI de Bourges for introducing me to the amazing field of system and network security in the first place.

I would like to thank the researchers I had the chance of collaborating with including Jill Jermyn, Ahmad Seify, Moustafa Alzantot, Mani Srivastava, Vidyasagar Saidu, Tuan-Anh Le, Mehdi Javanmard, Laleh Najafizadeh as well as my labmates: Rui Han, Sriharsha Etigowni, Luis Garcia, Rui Han, Pengfei Sun, and Mingbo Zhang.

I would like to thank Kaustubh Joshi for his valuable input and guidance on the projects presented in this thesis and for welcoming me at AT&T for a summer internship.

I would like to address a very special thank you to my advisor, Saman Zonouz for his constant trust, guidance and for offering me so many great opportunities. I really look forward to pursuing our collaboration.

Finally, I would like to express my gratitude to my friends Mélanie, Ambroise, Cédric and Pierre-Henri for keeping in touch and for visiting me despite the kilometers that separate us; to my grand mother Manuela, for her love and for sharing with me the secret of her best recipes, to Françoise for her advice and support, and to my father Jean and my sister Thérèse for their unfailing support.

Dedication

For my grand-mother Manuela, my father Jean and my sister Thérèse, for their support and their love. For my mother Manuela and my grand-parents, Henriette, Joseph and Luis, whom I wish could have read those lines.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile devices and embedded systems have become ubiquitous in our everyday's life. They enable a constantly growing range of applications by leveraging new hardware features. Their popularity has been fueled by app markets that enable new features and use-cases via a simple app installation.

In order to facilitate third party applications development, mobile operating system architects adapted their solutions to the constraints of mobile devices, including their limited resources and their constrained user interfaces. Among these choices, they designed the operating system to facilitate data aggregation from various sources under standardized storages such as centralized user account databases, and user address books. They also developed unified interfaces to access data. While this approach has led to a great deal of convenience for the developers and user to organize and access information, it has also opened the door for abuses.

Access control to these unified storages is enforced through permission mechanisms. The most popular mobile operating systems on the market, Google Android and Apple iOS, both use installation time or runtime permissions mechanisms to regulate applications access to the device sensitive data. However, permission mechanisms have been proven inefficient: they provide coarse-grained permissions with no alternative but to comply with the permission request [76, 39].

While users greatly benefit from applications functionalities, they also consciously or unconsciously expect their phone operating system and third-party applications to handle their personal data the way they, the users, expect to, to serve only a specific functionality. Unfortunately, user data protections expectations are not always aligned with the phone manufacturers or application developers' intentions. This dissymmetry

boils down to the two following specific scenarios: either the application is malicious or the user misuses the application. On the one side, the completely open application store distribution system has suffered abuses from app writers. As any developer can submit their app to the market, the application markets are crippled with malicious applications. On the other hand, user mistakes are also a factor of data leaks. Common reasons usually include usability issues or missing features [92].

Several solutions have been proposed to protect user data on consumer devices. A first approach consists in providing security through compartmentalization, to protect different sets of data and execution environments depending on the sensitivity of the task they handle [8, 14]. Such isolation techniques have been proposed at different levels in the operating system software stack. Virtualization techniques especially have been widely adopted in industry over the last two decades. However, these solutions often conflict with smartphone limitations: current devices only have limited resources with constrained interfaces which makes a full isolation solution, such as a virtual machine a burden on both the device resources and the user interface. Also, such attempts usually go against the flexibility users expect from mobile devices where information is aggregated and fully accessible.

A second approach relies on information flow tracking solutions to monitor sensitive data flows. These solutions follow data flows from specific sources, where the sensitive data is accessed, to specific sinks, where sensitive data leaves the device. While this concept seems directly applicable for data protection, current designs and implementations are often limited by their coverage or precision. More specifically, existing information flow tracking solutions provide different levels of granularity. For example, HiStar [96] provides a system objects (file, processes, socket) information flow tracking system. While these systems allow to monitor interactions between the system objects they cover (e.g. processes, files), they are usually pessimistic about finer grained objects and treat them as higher level object. For example, each byte read from a string generated by a sensitive process will be considered as sensitive. Conversely, too fined grained information flow systems incur a heavy overhead that render them unusable

for real world use. Finally, all of these solutions require heavy system changes or application instrumentation and fail to adapt to current application distribution models where system changes or application changes are a luxury only available respectively to phone resellers and application developers.

Unfortunately, all above-mentioned solutions assume a very restrictive threat model that include the operating system in the trusted computing base of the solution. This trusted computing based has been proven a very loose assumption over the years. Mobile device operating system have large the code bases and are targets of choice for all kind of attacks. Considering these limitations, this thesis proposes to answer the following problems:

- *How can we leverage current information flow tracking system techniques to provide a usable data protection mechanism for users?*
- *How can we enhance current information flow tracking system techniques to design a lightweight and non-intrusive information flow mechanism that does not suffer from the limitations of current solutions?*
- *How can we reduce the trusted computing base of current data protection solutions?*

This thesis proposes to investigate three data protection schemes for mobile devices. We first propose a generic data protection scheme, named Swirls, that enhances existing information flow tracking solutions to protect user data depending on specific device usages. We then propose a new information flow tracking technique, called Metron, that provide data tracking for numerical values as well as the computation history for leaked values and does not require system changes. Last, we consider the case of a fully untrusted device and propose a hardware protection mechanism for medical point of care devices, called Medsen.

Swirls: enabling data protection through virtual micro security perimeters. Mobile devices are increasingly becoming a melting pot of different types of data ranging

from sensitive corporate documents to commercial media to personal content produced and shared via online social networks. While it is desirable for such diverse content to be accessible from the same device via a unified user experience and through a rich plethora of mobile apps, ensuring that this data remains protected has become challenging. Even though different data types have very different security and privacy needs and accidental instances of data leakage are common, today's mobile operating systems include few, if any, facilities for fine-grained data protection and isolation.

Chapter 2, presents Swirls, an Android-based mobile OS that provides a rich policy-based information-flow data protection abstraction for mobile apps to support BYOD (bring-your-own-device) use cases. Swirls allows security and privacy policies to be attached to individual pieces of data contained in signed and encrypted *capsules*, and enforces these policies as the data flows through the device. Unlike current solutions such as VMs and containers that create duplication and cognitive overload, Swirls provides a single environment that allows users to access content belonging to different security contexts using the same applications without fear of inadvertent or malicious data leakage. Swirls also unburdens app developers from having to worry about security policies, and provides APIs through which they can create seamless multi-security-context user interfaces.

To implement its abstractions, Swirls develops a cryptographically protected capsule distribution and installation scheme, enhances Taintdroid-based taint-tracking mechanisms to support efficient kernel and user-space security policy enforcement, implements techniques for persisting security context along with data, and provides transparent security-context switching mechanisms. Using our Android-based prototype (>25K LOC), we show a number of data protection use-cases such as isolation of personal and work data, limiting document sharing and preventing leakage based on document classification, and security policies based on geo- and time-fencing. Our experiments show that Swirls imposes a very minimal overhead in both battery consumption and performance.

Metron: a value-based information flow tracking system. Mobile devices are equipped with a variety of sensors that enable various useful applications. While it is desirable to grant applications access to these sensors in order to accomplish their legitimate functionalities, ensuring that the sensor readings are not leaked to other parties is a challenging problem.

Information flow tracking techniques have been proposed to detect malicious data flows. However, the existing solutions suffer from several usability and precision issues that hinder their adoption.

Chapter 3 introduces Metron, an information flow tracking framework to detect potential data disclosures. Metron leverages a new lightweight information flow tracking technique that enables flow detection based on tainted values rather than a shadow memory taint system.

Metron leverages an application sandbox mechanism to carry the analysis of a monitored application. Unlike previous solutions, Metron implementation on Android works as user-space application that do not require modifying either the operating system or the target application. While legacy information flow tracking solutions rely on the abandoned Dalvik VM, Metron is compatible with the latest Android Runtime (ART). Our experimental results show that Metron provides a good accuracy for flow detection compared to other state-of-art solutions. It reports less false positives than TaintDroid, and similar accuracy to BayesDroid while handling numerical values that BayesDroid can not handle. Moreover, Metron allows the investigation of data leakage without modifying the operating system or the target application while adding acceptable overhead.

MedSen: a hardware-based data protection scheme for point of care devices. Trustworthy and usable healthcare requires not only effective disease diagnostic procedures to ensure delivery of rapid and accurate outcomes, but also lightweight user privacy-preserving capabilities for resource-limited medical sensing devices.

Chapter 4 presents Medsen, a portable, inexpensive and secure smartphone-based

biomarker[1] detection sensor to provide users with easy-to-use real-time disease diagnostic capabilities without the need for in-person clinical visits. To minimize the deployment cost and size without sacrificing the diagnostic accuracy, security and time requirement, Medsen operates as a dongle to the user's smartphone and leverages the smartphone's computational capabilities for its real-time data processing.

From the security viewpoint, Medsen introduces a new hardware-level trusted sensing framework, built in the sensor, to encrypt measured analog signals related to cell counting in the patient's blood sample, at the data acquisition point. To protect the user privacy, Medsen's in-sensor encryption scheme conceals the user's private information before sending them out for cloud-based medical diagnostics analysis. The analysis outcomes are sent back to Medsen for decryption and user notifications. Additionally, Medsen introduces cyto-coded passwords to authenticate the user to the cloud server without the need for explicit screen password entry. Each user's password constitutes a predetermined number of synthetic beads with different dielectric characteristics. Medsen mixes the password beads with the user's blood before submitting the data for diagnostics analysis. The cloud server authenticates the user based on the statistics and characteristics of the beads with the blood sample, and links the user's identity to the encrypted analysis outcomes.

We have implemented a working prototype of Medsen through bio-sensor fabrication and smartphone app (Android) implementation. Our results show that Medsen can reliably classify different users based on their cyto-coded passwords with high accuracy. Medsen's built-in analog signal encryption guarantees the user's privacy by considering the smartphone and cloud server possibly untrusted (curious but honest). Medsen's end-to-end time requirement for disease diagnostics is approximately 0.2 seconds on average.

Chapter 5 concludes this thesis and opens the discussion to potential future work.

---

[1]A biomarker, or biological marker, generally refers to a measurable indicator of some biological state or condition such as human disease. For example, the overabundance of certain blood cell types or biomolecular levels may indicate an infection.

Chapter 2

Virtual Micro-Perimeter via Information Flow Tracking

1 Introduction

Mobile devices have become the primary platform over which data is consumed. This data often have a large variety of privacy requirements. Today, confidential corporate email and documents, sensitive health and financial records, private audio and video conversation streams, and media intended for public dissemination on online social networks all jostle for the user's attention on the same phone. However, data isolation facilities on current mobile platforms leave much to be desired, with few OS facilities for preventing sensitive data from mixing with data for public consumption and leaking to untrusted endpoints, either accidentally or maliciously, e.g., a sensitive corporate email accidentally forwarded over a public email provider, or a private picture being shared over social media.

Solutions do exist for specific use-cases. For example, to protect sensitive company data, enterprises often require their employees to use different phones for work and personal use, or to use variants of such a scheme, e.g., partitioning of a single device into virtual "work phone" and "personal phone" through OS [8] or CPU virtualization [14]. Even when BYOD (bring-your-own-device) is permitted, content owners control the access to protected content by allowing access only to curated apps that are known to enforce fixed protection requirements. For instance, corporate email can be accessed only through a special corporate email client, and movies, books, or music from a particular publisher can be read only through that publisher's app. In most cases sharing data between apps is either completely disallowed, or restricted using install-time or runtime permission-based schemes such as Android's default permission control, SEAndroid [80], or ASM [49]. When data protection cannot be guaranteed by any of

these approaches, mobile devices are simply prohibited. For instance, phones might not be allowed in secret corporate labs to prevent leaked pictures of prototypes.

Unfortunately, these solutions fall short along key efficiency and usability dimensions. For instance, the development and maintenance of isolation environments that access similar types of content can become costly and burdensome for content owners and users. Content owners have to develop, maintain, and update BYOD apps for the sole purpose of enforcing data protection when, otherwise, a third party app might have sufficed. These apps often live in silo'ed containers, unable to access services provided by the rest of the mobile platform for fear of information contamination, e.g., a corporate email app might require its own PDF reader. For users, these multiple-environment systems present a fragmented and often inconsistent user experience that increases cognitive effort. They also limit users' choices to install only the specific recommended apps to access the sensitive material. Different apps for doing similar tasks behave differently, and must be individually configured according to user preferences. In cases where system resources such as cameras, microphones, or even device location are concerned, container-based (non-system-wide) approaches fail completely. A corporation concerned with leakage of unauthorized pictures from a sensitive lab area cannot simply publish a BYOD app to disable the camera, and the user may choose to simply ignore the app completely.

We make the key observation that the above approaches are limited by a fundamental mismatch: while security and privacy requirements are a property of the *data* being processed, the above schemes define security in terms of the *actors*, i.e., environments, processes, and apps. Therefore, they either suffer from an explosion of actors (one for each data type that needs a different security context), or they suffer from low granularity [41]. We propose a contrasting approach to address this dilemma: one in which a security policy is directly associated with the data by the content owner and flows through the system as the data it is attached to is processed by various apps. In this model, the same app should be able to process data from different contexts, while still ensuring that each data owner's potentially differing security requirements are enforced. Such a scheme can prevent leakage by prohibiting mixing of data belonging to

different security contexts and benefit users, content-owners, and developers alike. It allows users to use any apps they want to access content. It allows content-owners from being forced to write and maintain curated apps simply to enforce security policies. And finally, it frees app developers from having to implement security policies, and enables them to focus on developing their apps.

To implement such a vision, we present SWIRLS, an OS-level solution based on Android that provides system-wide policy-based *data* isolation by facilitating and controlling the mixing of data from different contexts. An OS-level solution is necessary to prevent malicious apps from bypassing the security policies required by data owners. SWIRLS allows data to be packaged in micro security perimeters called *capsules* that can be securely and dynamically installed or removed from the mobile device through an authentication and certification protocol.

Each capsule lists initial sensitive source objects. These objects are either files or system objects that generate data (SSL sockets). Capsule data is subject to policies specified by the capsule owner that dictate whether mixing with data from other capsules is allowed, whether the data may leave a device, and whether it must be encrypted or password protected. Policies can be based on contextual information such as location, time, data provenance, or online accounts.

SWIRLS tracks the capsule data propagation through the phone and enforces policies on both the original capsule data as well as data derived from it as it is used by various apps. To do so, we extend existing techniques for information flow tracking (TaintDroid) to allow policy control by disallowing operations that result in data mixing or other policy violations. As a result, SWIRLS monitors app execution from the system for potential policy violation as third-party apps run under the constrains of installed data policies.

This chapter details the challenges faced in designing and implementing such a scheme. In particular, while information flow tracking solutions theoretically provide an ideal approach to track and enforce policy on a piece of data, several limitations of current implementations make their adoption difficult for this specific use case. In particular, the level of granularity achieved by an information flow tracking solution as

well it's precision are key factors to consider.

The level of granularity achievable by an information flow tracking solution determines the type of flow that can be supervised. For example, while HiStar's [96] confidentiality and integrity labels apply to high-level objects such as files, inter-process communications, and socket connections, other solutions such as TaintDroid or DroidScope achieve an intra-application variable-level granularity that provide a much more refined view of a data flow. This granularity defines the level of separation or proximity that can be achieved between two piece of data carrying conflicting policies. While in the first case, no data from conflicting sources can coexist in a same process, for example, a solution like TaintDroid can achieve this use case. The second limitation to take into account is the precision of an information flow tracking solution. For example, while TaintDroid supports variable-level information flow tracking, it is by default using a single taint per array of variable stored in memory. Moreover, it does not track data flow through native code [1].

These consideration are at the center of SWIRLS design. We answer this granularity issue by providing an hybrid information flow tracking system that provide multiple levels of granularity. Our first approach considers the ideal case of an application that handles data with conflicting policies without merging it at any point in the execution. We call these applications "managed" apps. If such an approach is not achievable, we propose two alternatives: either 1) modify the application, if the source code is available, and improve it to prevent policy violations, or 2) run the application under a more coarse-gained information flow control that provides a more rigid data isolation and uses time sharing to accommodate access to data carrying conflicting policies. We call this last category of applications "unmanaged" apps. This chapter explores these three approaches.

Contributions. SWIRLS enables a new model for system-wide and user-transparent data isolation on mobile devices by associating security policies with data and enforcing them as the data flows across multiple third-party apps. The technical contributions of

[1] Native code support is deactivated by default on TaintDroid for third party apps

this chapter are organized as follows:

- We propose a new OS abstraction called *capsule* that allows content-owners to encapsulate sensitive data and its corresponding policies in a signed and encrypted wrapper. We develop a security-verified framework for dynamic capsule definition, distribution and on-device installation.

- We identify how existing information tracking engines (specifically, TaintDroid) can be enhanced to provide data isolation. Our extensions include efficient support for synchronizing policy enforcement across kernel and user-space objects, creating context-specific views of persistent data using union filesystems, persisting policy information across executions in the filesystem and in system databases, and developing efficient schemes for context switching from one security context to another.

- We propose new OS API interfaces to help third-party developers develop managed apps that present data from different security contexts to users in a cohesive and policy-compliant unified view.

- We implement a fully-working prototype of Swirls (>25K LOC) on Android `4.1.1_r6`, and validate its proposed dynamic security protection and performance with real-world apps.

Swirls considerably raises the bar for sensitive data protection and prevents accidental data leaks by third-party apps. However, like other taint analysis-based solutions, Swirls is unable to defend against system compromises or malicious apps that leverage side channels or implicit flows for information leakage.

The chapter is organized as follows. Section 3 overviews the three main requirements of this scheme. Section 4 presents Swirls's threat model. Section 5 presents the concept of capsules. Section 6 details how we achieved the solutions requirements for the managed and unmanaged apps. Section 7 describes the evaluation results. Section 8 and Section 9 go over past related work, and discusses Swirls's limitations. Finally Section 10 concludes the chapter.

2 Motivations and Use Cases

SWIRLS is motivated by the following BYOD use-cases, where employees use or are asked to use their smartphones to produce, modify, and consume data with different contexts and protection requirements. The context can range from coarse-grained, e.g., a user's work data vs. personal data, to fine-grained, e.g., data belonging to a sensitive corporate account vs. a general corporate newsletter.

The user's access level to data may also have to change based on the user's role in the company, time of the day, location, or any combination thereof. For instance, a company's accountant may access the financial database only during the work hours and from within the company. System-wide enforcement of these protection requirements across all apps without fragmenting the user experience is challenging and requires system level support.

Employer-employee: enterprise sensitive data access. Rather than providing employees with a separate smartphone for corporate use, enterprises are increasingly looking to reduce costs, provide greater choice, and reduce multi-device clutter by allowing workers to use their personal phones for work related apps (and vice versa). Many solutions such as L4Android [57], VMWare MVP [14], and Cells [8], have been proposed to address this need. They use either processor or OS virtualization to create multiple "virtual phones", one for work and another for personal use, that run on the same physical device. Qubes [52] introduces virtual containers to separate multiple domains based on Xen virtualization with a strict separation between different contexts.

Such container-based solutions provide complete *isolation* of contexts to keep entities related to each context absolutely separate from one another. Such an inflexible fixed architecture does not allow any data transfer across the contexts (too restrictive), while it permits all data communication requests between apps that belongs to the same context (too permissive). To guarantee context isolation, almost all existing solutions duplicate a full subset of system resources, such as context-specific copies of the same app, content provider, or system service. Besides wasting phone resources, such duplication reduces usability by increasing cognitive load on users. The user has

to keep track of the contexts manually and switch between them explicitly through a mechanism such as a touch screen swipe. The same apps have to be manually installed and updated in each context. A change of user preferences in one context does not automatically transfer to the other context. A practical solution that provides a unified environment with a single set of apps and still isolates work from personal data would greatly enhance usability while also reducing resource overheads.

Employee-employee: corporate-level secure data exchange. At enterprises, mobile devices are commonly used to bring rich corporate-related content and documents to employees by the company managers or other employees. In practice, the employees often have various levels of security clearance or roles within the company. Therefore, they should be allowed to access only particular sets of corporate content with specific security levels. As a more generic case, an employee may decide to share a document with a subset of other employees, e.g., sharing a sensitive project-related pdf file with teammates. Traditionally, such data separation is enforced at a human level: the employees share the received sensitive contents only with other colleagues working on the same project. However, data leakage may occur due to human error or due to permissive corporate containers that may mix different corporate data.

A system-wide BYOD data protection would enable employees to securely publish the signed sensitive content along with the corresponding policies. The policy-authorized employees could then access data through standard third-party apps, e.g., an Acrobat Reader for shared pdf documents or an MP4 player for video. The solution would enforce the policies across the system, and deny certain unauthorized action requests such as an employee emailing a plaintext protected file to an out-of-context contact.

Currently, Android does not provide fined-grained functionalities to access and protect data from several contexts with specific security clearances. Solutions like Bluebox [16] provide specific security measures such as data encryption for corporate apps. However, such coarse-grained measures cannot effectively support secure data transfer use-cases where fine-grained security requirements need to be enforced.

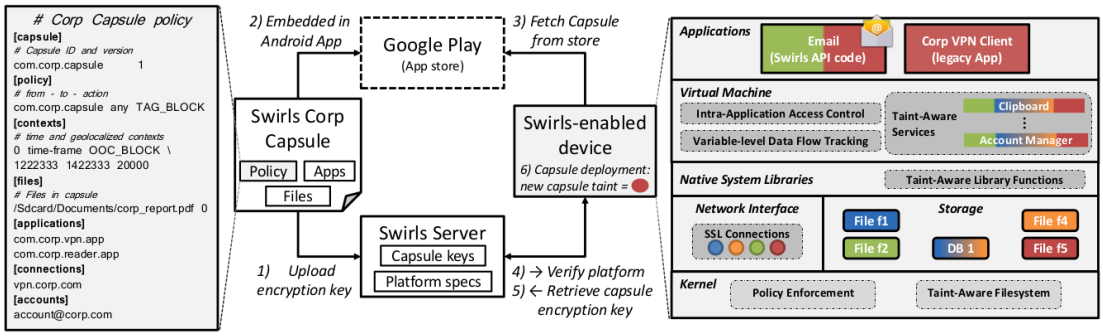


Figure 2.1: SWIRLS's High-Level Architecture

Additionally, the solution should support more complex time- and location-variant policies for when/where the sensitive data may be accessed. Specifically, the virtual micro security perimeter boundaries may change as a function of time and/or location. For instance, the attendees of a Federal conference may be asked access the sensitive shared contents only within the perimeter of the conference and during working hours. SWIRLS allows definition of temporal and location-dependent policies within capsules and controls the sensitive content movements. A capsule could define contexts and the corresponding policies based on time ranges and geographic locations when/where the capsule data is valid and can be accessed.

TempoGeoFence: limited device use enforcement. Some smartphone usage policies may be time- and location-variant. In other words, the virtual micro security perimeter boundaries may change as a function of time and/or location. For instance, the attendees of a Federal conference may be asked to not access the sensitive shared contents outside of the conference or off the working hours. SWIRLS allows definition of temporal and location-dependent policies within capsules and controls the sensitive content movements. A capsule could define contexts and the corresponding policies based on time ranges and geographic locations when/where the capsule data is valid and can be accessed.

3 Swirls Overview

Figure 2.1 shows SWIRLS's high-level architecture. SWIRLS's main objective is to facilitate data separation through deployment of virtual micro security perimeters that we call *capsules*. Each capsule is an encrypted and signed package that includes sensitive data and policies. These policies define how data should be treated when merged with data from other capsules. Figure 2.1's left block presents a simplified capsule policy. Every capsule is packaged and signed by its corresponding data owner who could be *i)* corporate admins who do not want the high-profile corporate data transferred from the device or mixed with other data; *ii)* app developers who intend to prevent uncertified parties from accessing specific files on the device; and *iii)* third-parties who may wish to control access to their data. SWIRLS verifies the integrity of the installed capsule signatures on the device before enforcing its associated policies. In the first case, the corporate admins protect their sensitive data, against outsiders, whereas in the second and third cases, the external parties (i.e., developers and third-parties) attempt to protect against curious and potentially careless device users.

Three major steps are required to enforce data protection in SWIRLS. First, for distribution and installation of a capsule, SWIRLS implements a secure protocol to prevent malicious capsule modification and/or interception attacks. Second, to protect sensitive data, SWIRLS employs dynamic taint analysis to keep track of the installed capsule boundaries while data moves within the system. Third, SWIRLS implements efficient mandatory access control at various data propagation points within Android to prevent unauthorized data accesses.

Capsule definition, distribution and installation. SWIRLS leverages a PKI to securely verify the origin of a capsule and perform a platform verification of the target device. Both steps are required to securely bind a policy to a target system's data. Section 5.1 presents SWIRLS policy definition and distribution scheme and evaluates the advantages of a data based policy against existing solutions.

Capsule boundary tracking. To guarantee data protection, SWIRLS keeps track of capsule boundary growth by tracing the sensitive data propagation starting from the capsule's source objects. SWIRLS deploys system-wide taint tracking techniques across various layers of Android to monitor data flow among the following system entities[2]: files, Android content providers, apps, system processes and services, account entries, secure socket connections, interprocess data exchanges, system service calls, as well as incoming network traffic. To retain the capsule boundary information across smartphone reboots, SWIRLS stores references to tainted objects for each capsule in a global database and keeps its information up-to-date whenever new objects are tainted.

Capsule policy enforcement. The capsule policies mandate how SWIRLS should handle access requests to different capsules' data throughout the system. SWIRLS's runtime policy enforcement uses the real-time information from the aforementioned capsule boundary database through a three level instrumentation of the Android framework. First, SWIRLS controls data accesses within the Linux kernel to ensure that the low-level capsule data propagation complies with the installed policies. This includes filesystem operations and inter-process communications among apps. Unlike previous work [35], SWIRLS's kernel-level support makes it resilient against malicious access control evasions through Java Native Interface (JNI) code segments. Second, SWIRLS instruments the Dalvik virtual machine (VM) layer with policy enforcement modules to control fine-grained access requests to variables within individual apps. SWIRLS's kernel-level enforcement is more lightweight than its Dalvik VM counterpart; however, it uses Dalvik layer enforcement for fine-grained access controls when a multi-context app includes data from different capsules simultaneously (SWIRLS's kernel-level implementation cannot distinguish different taints within an app). Finally, SWIRLS enhances and controls data interactions among several key system services, e.g., the clipboard service, that are accessed by multiple resources in the system and aggregate data from various sources.

[2]TaintDroid [35] does not support policy enforcement at any level, dynamic taint source assignment, and taint analysis across reboots, and among files, content providers, system services, accounts, syscalls, and network sockets.

Section 6.1 and Section 6.2 respectively detail the tracking and policy enforcement design for both managed apps, where several capsule's data coexist in a single app at the same time, and unmanaged app, that rely on a time sharing approach to access multiple capsules' data without breaking the policy.

4 Threat Model

Swirls's trusted computing base (TCB) contains the Android system (kernel, Dalvik VM, system services) and the Swirls server. For security, the server could be maintained by the company.

Swirls is primarily a system for preventing unforeseen mixing of data in apps. Swirls does not trust app developers to follow a data owner's desired data isolation policies. With perfect information flow tracking (IFT), apps would be completely untrusted. However, because of current practical limitations of data-flow tracking to support covert channels, our current implementation of Swirls cannot protect against malicious apps that actively circumvent isolation using covert channels and implicit flows. Fortunately, that requires deliberate effort from the developer, thus tagging them as bad actors if caught. In practice, for perfect protection against advanced malicious app, Swirls should be coupled with some mechanism to establish basic trustworthiness of app developers, e.g., through a developer certification program.

Swirls's primary use-case is to help the overwhelming majority of non-malicious apps that fail to meet user privacy and data isolation expectations in specific circumstances. For example, the Mac email client defaults to using an outgoing server associated with another account if the primary server associated with an account failed, thus allowing sensitive email on an enterprise account to be unknowingly sent via an untrusted public cloud-mail provider.

5 Capsules: Virtual Micro Security Perimeters

The next sections discuss Swirls's capsules format (Section 5.1), their distribution and on-device installation (Section 5.2) and discusses how they improve on existing policy

distribution approaches.

5.1 Capsule Architecture

Figure 2.2 presents SWIRLS's capsule policy grammar. Capsule identifiers, contexts, policies and objects are listed by section. Each object can be part of a specific context and be subject to a policy *action* when leaving its context. Figure 2.1's left-side box shows an example of a simplified capsule policy.

Upon capsule installation, the policy entries are read and enforced on the system. Specifically, each policy contains the following (possibly empty) entries. *ID:* A unique capsule ID for its corresponding data owner; *Apps:* The set of apps that SWIRLS initially marks with the capsule context. For apps that are not tagged by any installed capsule, SWIRLS launches them as taint-free initially. However, they may get tainted afterwards if they are destined by a data flow with a data context source. *Data:* The files and directories are linked with the capsule context and are considered as context sources. For instance, an app may come with its own sensitive files and directories that need to be protected. *Accounts:* The accounts in Android's Account Manager service corresponding to the target capsule. Once an app establishes a connection through a specific account, SWIRLS labels incoming data with the capsule context. *Connections:* The connections, e.g., SSL certificates, that SWIRLS considers as context sources. *Geo/time contexts:* Time intervals and geographic locations that determine when or where the capsule data should be considered valid. The capsule also includes an action (possibly NOP) that SWIRLS should take when leaving/entering the context, e.g., deletion or encryption of sensitive corporate data objects when the smartphone leaves the company location. The geographic locations are defined as circles (the center points to latitude and longitude information along with a radius). *Ruleset:* The set of policy rules that define how the mixing points among data from different capsules (contexts) should be handled. Data owners could define capsules with a set of possible policy actions in the case of any data mixing occurrence: the request may be allowed, allowed and logged, denied, or denied and logged. SWIRLS's policy enforcement engine prioritizes *deny* over *allow* in the case of conflicting capsule policies. We consider more

⟨*capsule*⟩ ::= ⟨*capsule-id*⟩ ⟨*context*⟩ ⟨*policy*⟩ [[⟨*file*⟩][⟨*application*⟩][⟨*connection*⟩][⟨*account*⟩]]
⟨*capsule-id*⟩ ::= **capsule** ⟨*cap-name*⟩ ⟨*cap-version*⟩
⟨*context*⟩ ::= **contexts** { ⟨*geo-context*⟩ | ⟨*time-context*⟩ }
⟨*policy*⟩ ::= **policy** { ⟨*capsule-id*⟩ ⟨*capsule-id*⟩ ⟨*action*⟩ }
⟨*action*⟩ ::= **ALLOW** | **DENY** | **ALLOW_LOG** | **DELETE**
⟨*file*⟩ ::= **files** { ⟨*path*⟩ ⟨*context-id*⟩ }
⟨*application*⟩ ::= **applications** { ⟨*package-name*⟩ ⟨*context-id*⟩ }
⟨*connection*⟩ ::= **connections** { ⟨*ssl-cname*⟩ ⟨*context-id*⟩ }
⟨*account*⟩ ::= **accounts** { ⟨*account-id*⟩ ⟨*context-id*⟩ }

Figure 2.2: Simplified Capsule Policy Grammar.

advanced system-wide policy consistency analyses outside the scope of this work.

Contrary to existing mandatory access control policies syntax, such as SEAndroid, SWIRLS policies are simple to read and to write. Solutions like SEAndroid realize fine grained access control to the cost of usually complex policies. Solutions like EASEAndroid[90] have explored approaches to automate the policy writing process for the Android platform but such a system still require human intervention and thus a good knowledge of the policy structure. By comparison our structure is simple enough so that even non-specialist users can express their data protection requirements in a few lines.

5.2 Secure Capsule Distribution

SWIRLS implements a secure capsule distribution and installation interface for nontechnical users. Figure 2.3 describes the capsule distribution and installation procedure. Upon the definition of a capsule, the capsule owner signs and encrypts it. The signatures and encryption keys are pushed to SWIRLS's remote server. Capsules are packaged in Android app files, and hence are distributable via the Google Play Store. The user downloads a signed and encrypted capsule, which is then installed by the SWIRLS system app on the smartphone. The installation consists of two steps. First, during a platform verification procedure, SWIRLS's remote server verifies the authenticity of the local agent on the system to ensure that capsule policies will be enforced correctly. Second, the SWIRLS system app verifies the signature, decrypts the capsule, using the keys obtained from the server, enforces the capsule policy and installs the capsule data.

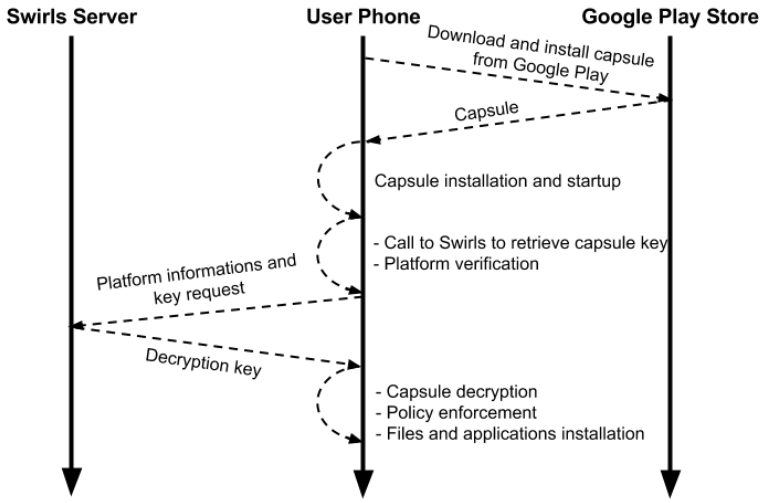


Figure 2.3: SWIRLS Capsule Installation via Google Play

Upon capsule installation, SWIRLS allocates a new and unique taint label and dynamically marks the capsule objects as *sources*. The objects can be apps and data files included in the capsule or sensitive data sources such as network connections.

As simple as this capsule distribution may be, it improves on SEAndroid policies distribution scheme since it does not require to be integrated into the application or the system development. The capsule embeds all the data sources that require protection along with optional files that are pushed to the device.

The capsule registration process is handled by the SWIRLS system application. This application keeps a database of all known capsules and tainted objects in the system. When modifications are performed (capsule installation or deletion), SWIRLS synchronizes the capsules policies and objects with SWIRLS's kernel security module policy cache. The taint database allows SWIRLS to keep track of the capsule boundaries over time and ensures the policy's persistence. At the device boot time, SWIRLS reads the capsule policies from the database and updates SWIRLS's kernel module accordingly through a communication channel. Once the installation is complete, SWIRLS allows the capsule's corresponding app to execute while SWIRLS traces the capsule's boundary throughout the system.

5.3 Capsule Context

SWIRLS enables the definition of location-based contexts defined within capsules. SWIRLS extends the Android `Geofence` API [6] for location-awareness. Every extended Geofence class object contains a capsule context ID, the GPS coordinates of the capsule's geographical context circle center and radius. The geographical fences trigger transition notifications to the kernel upon entry/exit into/from the context using the SWIRLS native library calls. To deploy the temporal contexts within the capsules, SWIRLS uses the Android `AlarmManager` API, where two managers mark the start time (context entry) and ending time (context exit). Like geographical fences, the temporal fences use the SWIRLS native library to trigger the temporal fence-based context switches at the kernel level. Upon the context change, the kernel will take the action defined in the capsule, e.g., to encrypt the sensitive corporate data after work hours.

6 Hybrid Information Flow Tracking Mechanism

SWIRLS uses capsule objects as *sources*, and considers any object from other capsules as data *sinks* where potential mixing of data from two capsules may violate the installed policies. Any capsule data mixing that violates a policy are blocked, irrespectively of the app since the data-flow monitoring and policy enforcement is driven outside the app, at the system level. The Android framework was modified in several ways to track capsule's data flows while running apps. However, existing apps do not need to be modified to run on top of SWIRLS. We therefore categorize apps in two groups:

Managed apps. They represent the ideal case where, by design, the app simultaneously accesses and processes data from different sources (capsules) that carry conflicting policies and does not merge their data during execution. Initially, apps run on managed mode as long as their execution does not lead to a policy conflict.

Unmanaged apps. They represent the apps that may not comply with the capsule policies at a variable-level flow tracking granularity. As a solution, we introduce an unmanaged execution mode that uses a stricter app isolation where capsule data is accessed using a time sharing approach.

Based on our observations, only a very small subset of existing applications can run in managed mode without accidentally merging data, including Android's system services that we had to modify to support the managed mode.

6.1 Managed Applications

This section first details the system changes made to the Android framework to support managed application and we introduce a new development paradigm and specific API methods that enable developers to enhance existing apps and ensure they can run as managed apps. The section that follows detail the case of unmanaged applications.

Capsule Boundary Tracking and Policy Enforcement

SWIRLS implements its capsule boundary tracking agents in the kernel and the Dalvik VM. It uses a kernel Linux Security Module (LSM) as a reference monitor for the capsule policies and it enhances the Dalvik virtual machine to gather intra-app semantic information, e.g., variables. Some other important changes were made to the system to ensure that system components would isolate capsule data by design.

LSM reference monitor. The reference monitor is implemented as a LSM that provides a character device in order to allow data propagation reporting among SWIRLS components within and outside the kernel. The LSM checks if the propagation reportings are compliant with the policy and return the result to the component.

Within the kernel, SWIRLS LSM makes use of the file system hooks to keep track of file accesses. We instrumented LSM hooks, namely `security_dentry_open()`, `security_file_permission()` and `security_path_unlink()`, to inform components about the capsule they are accessing and to update SWIRLS's capsule database if a file gets deleted. Similarly, the `read` and `write` operations propagates the capsule information to the variable receiving data or the file written to.

The Dalvik virtual machine uses this interface to report capsule object accesses in applications to the LSM via `ioctl` syscall.

The capsule boundary tracing and reference monitor implementation in the kernel

have two major advantages: they provide a centralized and privileged domain to monitor system-wide capsule data flow, and enables SWIRLS to keep track of native code operations on OS objects.

Dalvik variable-level Information Flow Tracking. SWIRLS relies on TaintDroid's variable-level taint tracking framework to keep track of information flows within managed apps. However, several changes were made to adapt the existing framework: 1) we substituted the statically defined taints by the capsule identifiers, 2) we report flows to the reference monitor on any data mixing involving different capsules, 3) we define a new set of sources and sinks: the capsule objects. The variable-level IFT mechanism is used in particular to detect incoming capsule data from SSL sockets and user accounts related sockets.

As a matter of fact, the majority of apps (according to our studies, 85.9% of 217 top apps) request the Internet access permission to open network sockets. SWIRLS treats each secure network connection as a potential source based on the remote endpoint indicated in its SSL certificates. We limit the incoming data sources that SWIRLS considers to secure sockets because it requires a validation of the endpoint through SSL/TLS certificate verification.

SWIRLS looks up the capsule database for each SSL/TLS connection during the handshake phase. SWIRLS extracts the *common name* [45] that matches the fully qualified domain name. SWIRLS compares the common name to the installed capsule connection objects, and labels the SSL/TLS socket with the taint of the matching capsule. The connection is not tracked if no matching capsule is found. SWIRLS instruments the socket's read and write calls to respectively mark data contexts or check policies to block unauthorized outgoing flows, e.g., different outgoing data and socket contexts.

SWIRLS enhances the `NativeCrypto_SSL_do_handshake()` function in Android Apache Harmony framework to extract the certificate common name and attach a context corresponding to the matching capsule, if any, and instruments the `NativeSSLSocket_read()` function to tag the incoming data its corresponding capsule once the connection is established.

TaintDroid also implements a variable level IPC mechanism that we reuse between managed apps and the three system components that follow.

System accounts. According to our experiments, 13.0% of apps read and/or write *all* account credentials[3] (Section 7.4). SWIRLS solves the lack of discrepancy among the accounts through its app-level context analysis that is aware of the account semantics. Monitoring such activities from within the kernel (lower overhead) is unfeasible as it requires high-level semantic information.

Content providers. SWIRLS instruments Android `ContentProvider` to store the context information along with the content of each individual entry. Many Android components, such as Android `contacts list`, use content providers for their data storage, and SWIRLS's instrumentation turns them into context-aware entities. Thus, SWIRLS can save the user's contacts from different contexts within the same contacts list database on the phone.

System services. The Android permission system restricts app accesses to system services, such as clipboard, account manager, hardware devices and sensors. However, a system service would mix data from different capsule. To provide capsule data isolation inside system services, SWIRLS leverages the Android multi-user support (since Android 4.1.1_r6) to separate capsule data.

Swirls API

To facilitate managed app development, SWIRLS provides a capsule-aware API that allows developers to query variable taints and check the capsule policies before variable value assignments. The API contains two main methods: `gettaint(Object o)` returns the taint of a specific variable or resource (file, socket, account, variable); `isAllowed(Object o1, Object o2)` checks if data flow from `o1` to `o2` is compliant with the installed capsule policies. It is noteworthy that there is almost no sensitive

[3]Using the Android `AccountManagerService` after acquiring appropriate permissions, i.e., `GET_ACCOUNTS`, `USE_CREDENTIALS`, `AUTHENTICATE_ACCOUNTS` and `MANAGE_ACCOUNTS`.

system information leakage as the result of API function calls to apps because taint IDs are opaque identifiers assigned to a capsule at its installation time and cannot be created or manipulated by apps.

We use the case of Android's AOSP email client app as a driving example. As a basic requirement, we modified the email client such as it accesses emails from different sources simultaneously while keeping their corresponding data separate in the system. Moreover, SWIRLS's API enables further modification of apps to make them capsule-aware in order to *i)* provide an enhanced context-aware UI, e.g., different colors for different emails based on their source; *ii)* change behavior based on capsule policies, such as blocking the transfer of an email between two recipient belonging to two different capsules; and *iii)* provide more user-friendly policy violation responses, such as a notification before sending a email going against the policy out. Our SWIRLS-enabled email client only required 320 LOC changes to a 183,076-line app to turn the original app into a capsule-aware (managed) app. This section details how we modified the Android system framework to handle this scenario and we present how SWIRLS enables the development of managed apps through a simple API and a dynamic code analysis technique.

Overtainting Resolution in Managed Application

Overtainting issues are likely to happen in applications running in managed mode. Their origins are twofold: either the application purposely mixed data from different capsules because its execution requires it or the taint tracking solution did generate a false positive, due to a variable sensitivity limitation, e.g.: for example, TaintDroid use a single taint per array.

Tracking down the changes required to avoid an accidental mixing between capsule in an application code can be a delicate task. As a matter of fact, the execution point where a policy violation was detected does not correspond to the root cause of an inexact flow detection. As a debugging tool to help the developer to track both cases, we enable a dynamic tainted instruction tracing feature in TaintDroid and use a backward slicing analysis on the generated trace to identify the root cause behind conflicting flows.

We successfully used this technique to instrument the capsule-aware email application.

6.2 Unmanaged Applications

For unmanaged apps that do not support an intra-app data flow policies, SWIRLS implements an app container mechanism based on Linux namespaces. This mechanism ensures the application handles only one capsule at a time and that different capsules are accessed sequentially. When an unmanaged app is launched, the user choses which capsule to process next via the interface. A policy violation inside an application becomes impossible. However, IPC between apps processing conflicting capsules remains a problem. This section details our approach to address this case.

Unmanaged Applications Compartments

Through kernel-level taint tracking, SWIRLS ensures that unmanaged apps stay either taint-free or single-tainted. When launching an unmanaged app, SWIRLS marks the app process with a capsule, by updating an added *process_taint* element within the kernel's `task_struct` structure, if requested by an installed capsule. The `startViaZygote()` method in the `android.os.Process` class queries SWIRLS kernel module for the taint (possibly null) for the app, and passes it as an extra parameter to the Zygote process. Consequently, the Zygote process assigns its child, i.e., the target app, with that taint at the fork point.

SWIRLS uses mount namespaces on the forked Zygote processes to keep track of tainted data generated by the launched app. SWIRLS assigns a directory to each capsule in the system, e.g., `/data/swirls/1`, `/data/swirls/2`, etc. During an unmanaged app launch, SWIRLS *i)* maintains separation among processes by creating a specific mount namespace for the launched process with the taint `tid`; and *ii)* sets up a capsule-aware filesystem directory tree via mounting a stackable Unionfs [71] link between `/data/data` in read-only mode and `/data/swirls/tid` in read-write mode. It is noteworthy that Android apps typically store their data under the `/data/data` directory, and `/data/data` contains app libraries that do not need to be replicated for

each taint. Using the copy-on-write mechanism, the Unionfs mounts in SWIRLS minimizes the amount of data replication from the initial installation data folder while the app may run with several taints. Whenever a launched app with a specific taint `tid` requests a shared data read from `/data/data`, e.g., a library file, the data is actually read from `/data/data`, but the write requests cause data-writes in `/data/swirls/tid`. Since `/data/data` needs to be shared and accessed by several processes, SWIRLS mounts the corresponding directories using the bind option.

Inter-processes Policy Violation Resolution

Our manual investigation of 270 real Google Play market apps showed that such IPC-based policy violations, especially through inter-app and app-service communications, occur frequently. Hence an access control solution is needed. Binder-based policy enforcement in SWIRLS is effective for all direct calls between processes with single or no taint; however, managed apps with more than one taint bypass such kernel-level enforcement and are addressed using the more fine-grained Dalvik layer mechanisms.

SWIRLS prevents unmanaged apps processing conflicting capsule data from exchanging data through the Binder IPC. SWIRLS enforces the capsule policies within Android's Binder IPC mechanism. SWIRLS leverage a Binder SEAndroid LSM hook [81] to evaluate the capsules associated with the unmanaged apps performing a Binder call. If the policy does not allow the call, SWIRLS transparently restarts the target app in the right context. This restart process leverages the Android `LowMemoryKiller` feature, initially intended to silently kill the apps over-consuming memory, to kill the target application and prevent the unauthorized IPC data exchanges. Processes that run system services[4] cannot be simply restarted; SWIRLS whitelists the processes and implements the per-service capsule-aware data flow control mentioned above. Section Section 7.3 provides an evaluation of the performance and usability cost of this approach.

SWIRLS also enhances the Android activity manager that is a system service used for inter-app component call resolution. It is a critical component to monitor since

[4]Such as `system_server` and `surface_flinger`.

Table 2.1: Android Framework Enhancements by SWIRLS

Component	**Changes or Added Feature**
Linux kernel (Grouper 3.1.10)	Policy cache, interface through IOCTLs, LSM hooks for files, binder hook
System services: ActivityManager, Clipboard, Accounts	Check caller context on requests
SSL native interface	Assign variable context according to data received from socket
Taintdroid (AOSP v.4.1.1_r6)	Data sources redifined, policy enforcement support on capsules' data merging
Zygote and Process class	Context assignment on process fork
Content Provider framework	Store data context for each database table row system-wide

the indirect calls relayed by the activity manager would not be visible by SWIRLS's Binder policy enforcement module. SWIRLS implements an enforcement agent within the activity manager that monitors the call chain as well as all the app launches and activity switching requests by the running apps. If a call violates the capsule policies, SWIRLS prompts the user for a response action, e.g., block.

Table 2.1 resumes the changes made to the Android Framework.

7 Evaluation

We implemented SWIRLS on Android 4.1.1_6 ($> 25K$ LOC C/C++/Java). Table 2.1 summarizes the system components enhanced by SWIRLS modules. Figure 2.4 illustrates the components that store SWIRLS meta data and policies (gray boxes) as well as the components that we modified to implement SWIRLS functionalities (white boxes). Our empirical evaluations were on a Nexus 7 tablet device to answer the following questions: *i)* how much performance overhead does SWIRLS cause compared to the vanilla Android system? *ii)* does SWIRLS detect unauthorized capsule boundary mixing and enforce the dynamically-installed policies accurately? *iii)* are SWIRLS's capsule definition/distribution/installation, and system-wide policy enforcement usable in practice? and *iv)* does SWIRLS realize the BYOD use-case scenarios successfully?

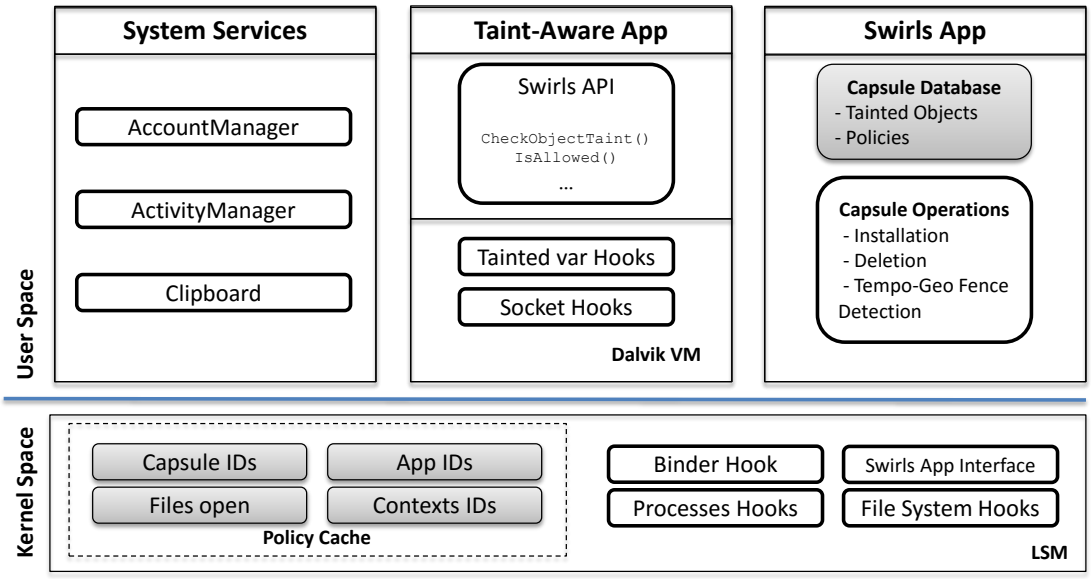


Figure 2.4: SWIRLS's System Components

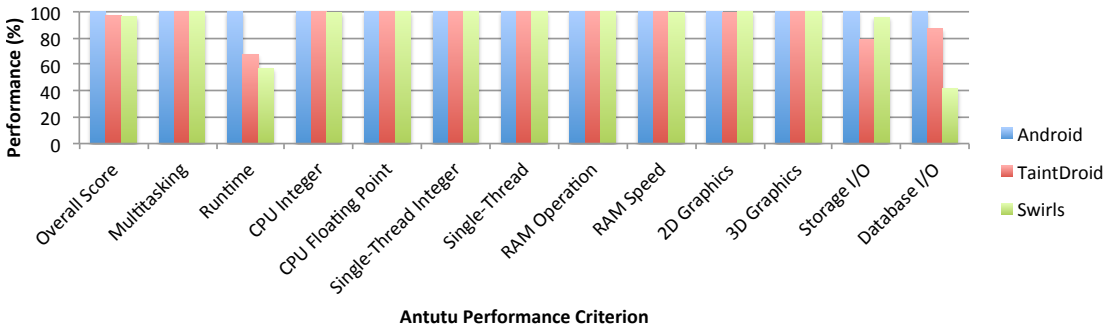


Figure 2.5: Antutu Benchmark v5.7.1 Performance Results

7.1 Swirls Performance

We measured SWIRLS performance overhead on Antutu benchmark [9] that gives performance scores for various criteria such as database IO, graphics, etc. Figure 2.5 shows SWIRLS's performance compared to Android vanilla (the base) and TaintDroid `4.1.1_r6` for the `userdebug` build. SWIRLS's overall performance is 96% of the base Android's performance, where TaintDroid's is 97%. The runtime performance score corresponds to the Dalvik VM, where most of the fine-grained (relatively high overhead) capsule boundary tracking and policy enforcement occurs. SWIRLS's relatively low database performance (41%) is due to its fine-grained instrumentation of the content provider framework. SWIRLS stores and retrieves the context for each data row

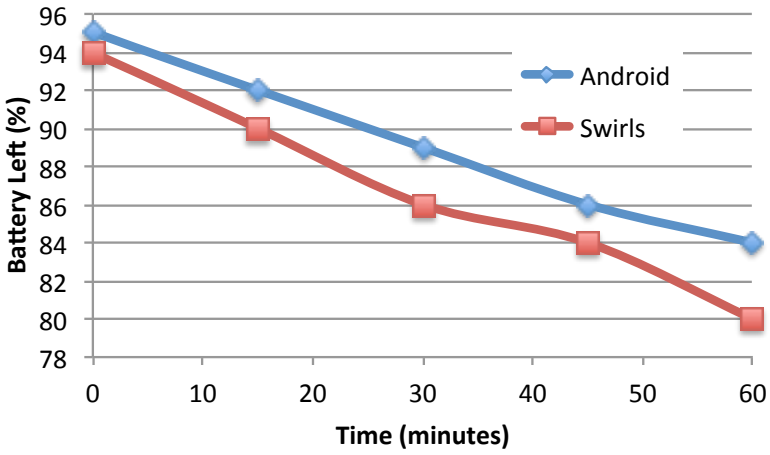


Figure 2.6: Runtime Battery Consumption

entry of the SQLite database. SWIRLS checks on every database `INSERT` or `SELECT` if the target table/row is currently tainted and, if so, whether the access request violates the installed capsule policies. Table 2.2 shows SWIRLS's runtime memory usage averaged over 15 apps that run by default at Android startup. Compared to the base Android, SWIRLS causes 9.5% memory overhead per app vs. 9% by TaintDroid. Finally, Figure 2.6 shows how SWIRLS affects the device's battery lifetime. SWIRLS drains the battery 3.8% more than the base Android after one hour of use, which is promising given SWIRLS's practical usability.

7.2 Capsule Boundary Evolution

To validate the need for a capsule boundary tracking and policy enforcement, we analyzed the top-10 most-used Google Play market apps to determine whether they merge sensitive data from different sources. Table 2.3 and Table 2.4 show the results of using SWIRLS's capsule boundary tracking engine. We considered established socket connections as separate data sources. We observed 2,819 (117 unique) data mixing incidents (Table 2.3) that were mostly caused by almost half of the apps (4th column). Most of the mixings occurred at the filesystem level (2,178). We also noticed a few cases where the app merged data from different individual sockets destined to the same institute, and hence did not violate SWIRLS's data leakage policies. We manually investigated the `com.android.vending` app, which mixes data from a large number of sockets;

Table 2.2: Memory Usage of the 15 Vanilla Apps Running by Default

Case	**Android**	**TaintDroid**	**Swirls**
Memory usage (KB)	502,612	545,664	548,168

Table 2.3: Data Policy Violations (DM: data mixing; UDM: unique DM; RA: responsible apps; UDMP: unique DM per process)

Object type	**#DM**	**#UDM**	**#RA**	**#UDMP (top 3)**
Files	2178	85	4	android.vending 71; system_server 1; whatsapp 14
Strings	536	26	10	android.vending 5; whatsapp 10; pandora.android 3
Variables (bool, double, int, array)	105	7	3	android.vending 3; system_server 1; whatsapp 3

however, all those sockets were connected to different servers at Google and the transferred data was mostly Google account parameters. Using certificate-based treatment of sockets, Swirls was able to correctly mark all of those connections as a part of single capsule that resulted in a single-context app with no policy violations.

Table 2.4 shows selected results for the top four popular applications. The second column shows whether the data mixing occurred due to a background app process (Facebook and Flashlight) or the user's action on the app's GUI. Third and last columns show, respectively, the points dynamically marked as data sources by Swirls and where data from different sources mix. The large number of mixing incidents by the current apps necessitates deployment of policy-based data isolation solutions like Swirls in practice.

Figure 2.7 illustrates the dynamic growth of two BYOD capsules that represent two different email accounts (personal and professional). The vertical axis lists different application objects, and the horizontal axis shows the sequence of various user activities over time. For instance, the database entry (object c; third line on vertical axis) is marked with the personal context once the user sets up the first account (A1 on the horizontal axis). The details of the object c is explained by message box on the figure. For each account, the capsule data sources consist of a single object entry: the SSL/TLS common name of the IMAP server. Swirls marks the data received

Table 2.4: Observed Data Mixing Incidents

App	Responsible action/entry	Data sources	Data mixing points
Facebook	Service (background)	Accounts and sockets (*.facebook.com, a248.e.akamai.net, *.xx.fbcdn.net)	App files, sockets, strings
Play Store	Adding account (UI)	Accounts and sockets (*.google.com)	App files, sockets
Pandora	SignUpActivity (UI)	Accounts and sockets (tuner.pandora.com)	Strings
Brightest LED Flashlight	Main activity (UI)	Accounts and sockets (*.flurry.com)	Flurry agent framework file and socket, binary blob

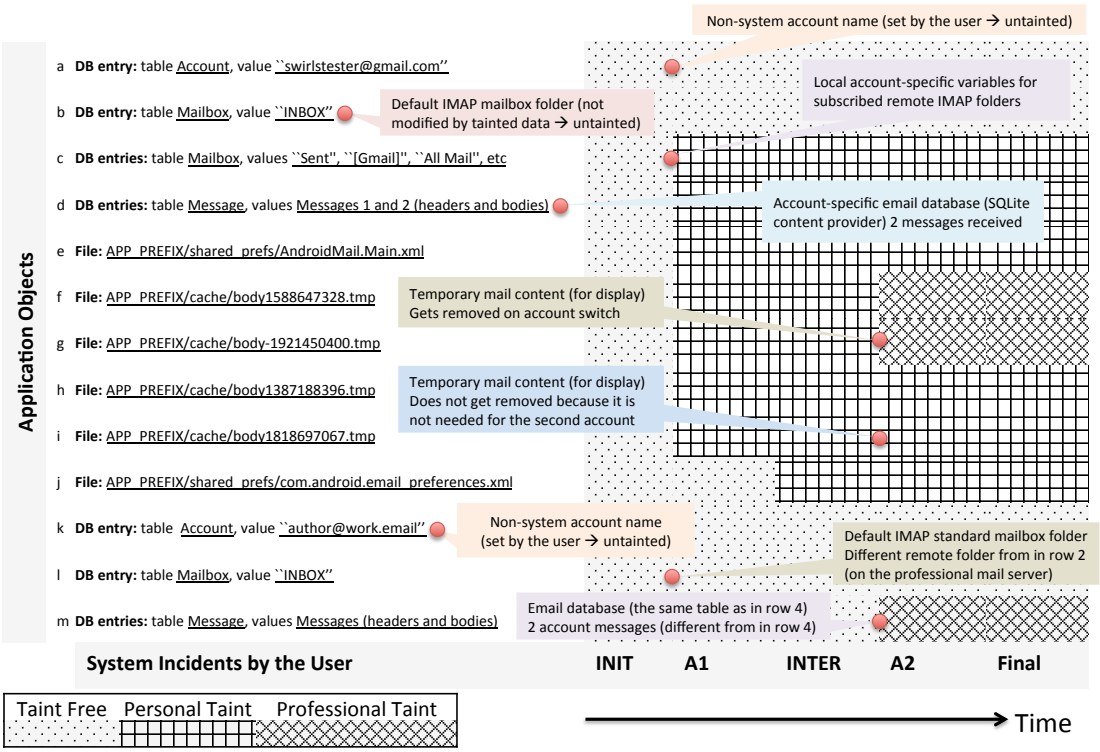


Figure 2.7: Capsule Growth for the Email Client Use-Case. (**INIT** = Contexts at Initial State, **A1** = Contexts After First Account Setup, **INTER** = Contexts After First Account Use, **A2** = Contexts After Second Account Setup, **FINAL** = Contexts at Final State, `APP_PREFIX=/data/data/com.android.email`)

Table 2.5: Context Switch (seconds) for Policy Violation Scenarios

Scenario	**mail & exchange**	**mail & acore**	**K9 & acore**	**Avg.**
Time	0.29	1.55	2.09	1.31

from the SSL/TLS with a specific capsule context depending on the end-point SSL certificate. The enhanced email app checks the context information during the email sending procedure to stop the process if the sensitive data transfer conflicts with the installed policies (as in Figure 2.9). The email body, the recipients' email addresses and possible attachments' contexts are inspected before allowing a mail to leave the device. SWIRLS's data flow policies enable the app to reuse the same file name in different contexts. For example, the temporary file `body1588647328.tmp` appears to be created while consulting the first account, then deleted while switching accounts and created again for the second account.

7.3 Capsule Policy Enforcement

Unmanaged app. We evaluated SWIRLS's time requirement for BYOD applications in the case of unmanaged apps where a policy violation causes a data context switch. Table 2.5 shows the results. We created multiple scenarios that all eventually led to policy violations; *i)* an *unmanaged* Android's default mail client tries to use an Exchange (professional) service while it is currently running in a different (personal) context; *ii)* the Android's default mail client tries to access the contact provider (the `acore` process) for an entry from a different context; *iii)* similar to (ii) but using a third-party K9 mail client app. The table shows the time requirement from the access request denial (e.g., Binder call rejection) because of the policy conflict until the launch completion of the process with the new context. The context switch takes approximately 1.31 seconds that is reasonable for practical usages.

Managed app development. We analyzed the default Android email client app by reading its source code and using SWIRLS's capsule boundary tracking to determine the main challenges in rewriting legacy apps and turning them into context-aware managed

apps. The managed apps should use the SWIRLS API (Figure 2.4) to handle multiple BYOD contexts simultaneously and prevent intra-app capsule data mixing such as an email forwarding between two different context accounts. We observed that data mixing occurred because of only a few points in app's source code. The mixings occurred in files (local mail account database entries and system service files), string variables, and the content provider entries in the app's data directory. We modified the app accordingly with a small amount of effort; the updated app was context-aware and did not mix data from different context accounts. Additionally, it was able to enforce more complicated capsule policies in SWIRLS, e.g., encrypting work emails after the work hours. Figure 2.8 (left block) shows the user interface of the updated app where emails from each BYOD context are color encoded. Overall, we changed 13 source code files adding/removing 320/5 lines of code within the app. The changes included *i)* modification to remove or duplicate the variables that get doubly tainted; *ii)* policy checking at sensitive places in the code, e.g., the `sendMessage()` function; and *iii)* UI and response action implementations to raise a notification (Figure 2.9) and ask the user to modify the mail to avoid the policy-violating data leak.

7.4 Realized Smartphone Use Cases

Employer-employee: enterprise sensitive data access. Most of the current apps have limited or no support for BYOD use cases. Based on our analysis of 285,457 top free apps on Google Play store, many apps access universally accessible shared spaces such as content providers and external storage, where data from different contexts could mix leading to capsule policy violations. In particular, 245,315 apps (85.9%) requested Internet access; 133,133 (46.6%) asked for access to external storage; 37,153 (13.0%) requested access to account information or credentials; and 13,638 (4.7%) requested access to contacts. Our findings through dynamic app analyses (Section 7.2) and the static investigations above demonstrate actual and potential unauthorized mixing of data from different contexts. This hinders the deployment of a secure environment where various context data interactions should be regulated system-wide. For instance, any two applications with shared space access permissions could set up a bi-directional

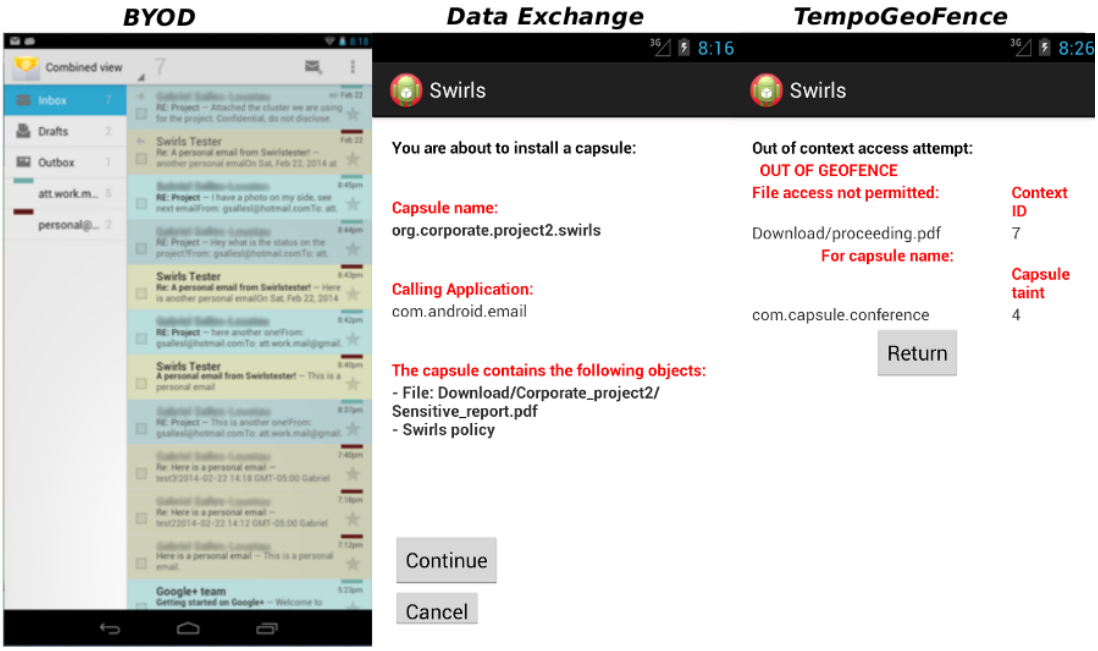


Figure 2.8: Swirls-Enabled Smartphone Use Cases

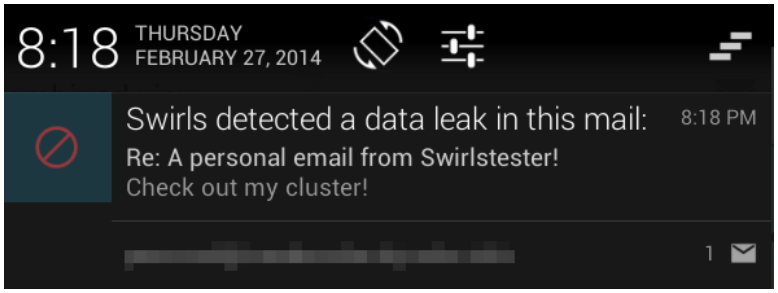
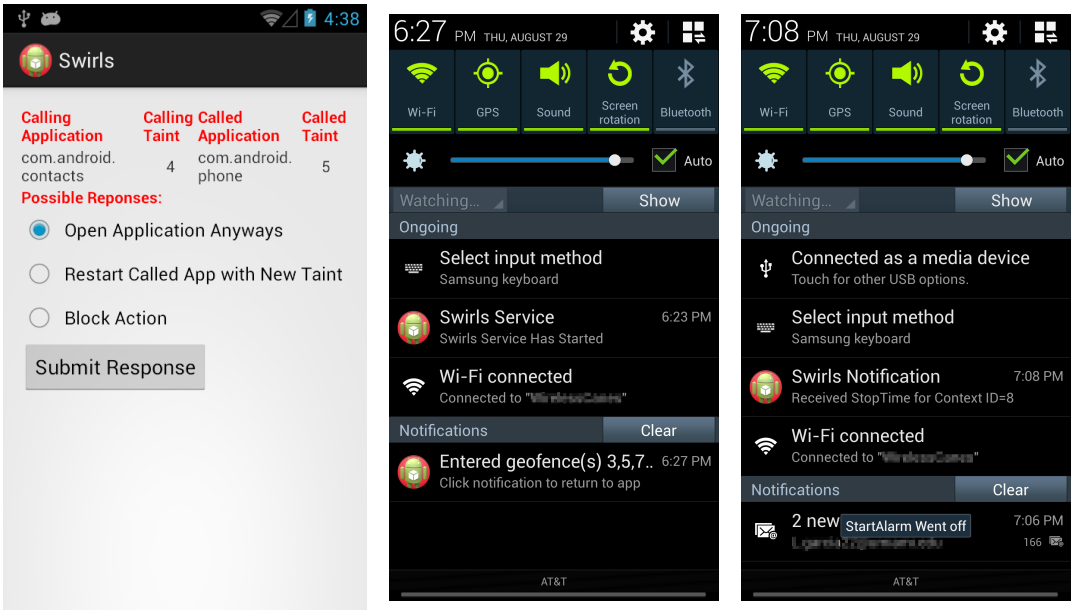


Figure 2.9: Violation Notification in a Managed App



(a) Policy Violation (b) Geofence (c) Geofence

Figure 2.10: SWIRLS User Interface

communication channel resulting in unauthorized data interactions.

SWIRLS facilitates an efficient way to realize a multi-context BYOD device usage experience. Capsules facilitate system-wide context/persona definition and data isolation. Our developed context-aware email client (Figure 2.8) shows how a mail containing an attachment downloaded from a professional account and forwarded to a professional contact eventually gets blocked during the send process. We uploaded an anonymous demo of SWIRLS's BYOD use case in [84]. In the event of a policy violation, SWIRLS asks the user for an action to take. Figure 2.10a shows a screenshot of SWIRLS preventing an app from launching that is already running under a different context. In the case of policy violations caught in the Binder driver, SWIRLS silently restarts the target component process in the new context (Table 2.5).

Employee-employee: corporate-level secure data exchange. We evaluated SWIRLS for the BYOD data exchange use-case where employees of the same company could share data with another subset of employees, e.g., who are in the same sensitive project team. SWIRLS enables data owners to export policies for sensitive files across devices. The SWIRLS server accepts requests from registered systems/users and creates capsules on-the-fly. In our use-case, the employee initially downloaded a sensitive

project file `sensitive-report.pdf` (incorporated into the corresponding capsule) from an email from his colleague at the same company. Figure 2.8 (middle block) shows what the employee observed upon opening the email attachment. This step was followed by the capsule installation and legitimate data access that was allowed by SWIRLS. Later, the employee intended to send the file to another colleague who was not a member of the project. Consequently, SWIRLS denied the email send request as it would have violated the installed capsule policies. We also implemented a temporal and location-based context switching app. Figure 2.10b- Figure 2.10c and Figure 2.8 (right block) show the SWIRLS notification when entering a geographically constrained context. Based on our experiments, this use-case drains the battery faster compared to other case studies mainly because of periodic GPS pooling by SWIRLS to check and enforce the installed capsule policy.

7.5 Comparison with Existing Solutions

We now compare the security protection provided by SWIRLS with other existing most related solutions.

TaintDroid [35]. SWIRLS significantly improves TaintDroid's capabilities through its policy enforcement agents across the system, dynamic taint source assignment, and taint analysis across reboots, introducing several new data sources and sinks (files, content provides, syscalls, and network sockets), IFT support of apps with native code, a verified dynamic capsule definition/distribution/installation framework, system-level API for context-aware app development, and isolation of tainted data processing in system services, e.g., binder, clipboard, account manager, device managers.

TaintDroid uses extended file attributes to store taints. Therefore, apps could change their taint information, that TaintDroid (and not apps) should maintain, at will by updating their file attributes. SWIRLS takes an alternative kernel-level file taint tracking approach for two reasons. First, SWIRLS should not allow app developers to manipulate their app's file taints (it is noteworthy that the app developers are often

different from the data owners who define data policies in SWIRLS). Second, the kernel-level support allows SWIRLS to maintain a centralized real-time system-wide database of capsule boundaries (the list of tainted system objects) rather than distributed labels on individual files. Centralized capsule boundary maintenance enables SWIRLS to accelerate enforcement of some policy types, e.g., "remove all professional data after working hours". For instance, in TaintDroid's architecture, this would require sweeping the whole filesystem, whereas in SWIRLS, removing files within the "professional data" table of the capsule boundary database would suffice.

Asbestos [31] and Histar [96]. SWIRLS's initial architecture for tainted system services (that run Android services) followed Asbestos's design; however, we had to totally redesign it due to resource limitations in smartphones. Initially, SWIRLS duplicated the `system_service` process threads for every taint, so IPC communications among tainted apps and system services would comply with the capsule policies. We dropped this implemented approach due to the added complexity of concurrent threads accessing hardware-dependent services and its high overhead in the worst case (#threads = #services*#taints). In our current implementation, we have instead modified the system service implementations, e.g., Account Service, Clipboard, and Activity Manager, to be taint-aware and keep different taints separate. This improved design reduced SWIRLS's performance overhead significantly.

Histar provides IFT using initially fixed Asbestos labels on kernel objects, e.g., threads, containers, and devices. Histar policy uses hierarchical *fixed* integrity levels associated with *categories*, i.e., data operations such as read and write. SWIRLS allows dynamic capsule policy updates, and enables enforcement of more expressive generic policies that are not simply deployable using hierarchical integrity levels. For instance, consider the following policy rules for contexts A, B, and C: A can flow to B and C; B can flow to C; C can flow to A but not B. There is no straightforward hierarchical integrity level assignment that would allow the information flow above. Additionally, SWIRLS allows simultaneous and policy-compliant multi-taint data access by the same

process using its fine-grained IFT support. HiStar does not support multi-taint processes.

SE-Android [80] and Knox [75]. SE-Android adapts SE-Linux mandatory access control (MAC) policies for Android platform to protect system components such as system services, Zygote-forked processes and third-party apps (it does not deploy different policies for individual apps). AOSP [7] strongly encourages the users not to add, delete, or modify any SE-Android policy. This could result in an inflexible architecture with too generic and coarse-grained policies that are not customized for individual apps.

SE-Android policies mandate the object domains that each app can access. The following SE-Android policy [80] marks the folder `/data/data` as `app_data_file` type, puts *all* third-party apps in the `untrusted_app` domain with the `app_data_file` type, allows them to create files and directories on SD-card, and lets third-party apps read files with `app_data_file` type (does not distinguish data from different apps, so an app can access other apps' data) and allows to exchange binder calls with any other app.

```
##### sepolicy/file_contexts #####
## App sandboxes
/data/data/.*           u:object_r:app_data_file:s0
##### sepolicy/seapp_contexts #####
user=app_* domain=untrusted_app type=app_data_file \
levelFromUid=true
##### sepolicy/apps.te #####
type untrusted_app, domain;
app_domain(untrusted_app)
## App sandbox file accesses.
allow appdomain app_data_file:dir create_dir_perms;
allow appdomain app_data_file:notdevfile_class_set \
create_file_perms;
## SDCard rw access.
bool app_sdcard_rw true;
if (app_sdcard_rw) {
 allow untrusted_app sdcard:dir create_dir_perms;
 allow untrusted_app sdcard:file create_file_perms;
}
## Perform binder IPC to other apps.
```

```
binder_call(appdomain, appdomain)
binder_transfer(appdomain, appdomain)
```

Knox [75] deploys stricter fixed SE-Android policy rules for corporate apps to support BYOD, and similar to SE-Android, requires initial system-wide data labeling. Using Knox for BYOD, employees need to use separate email apps to check their personal and professional emails. SWIRLS provides more usable solution through dynamic secure policy definitions and distribution, and enables user-transparent data access control through its unified user interface for various contexts. Additionally, SE-Android policies are app-based and monitor control flow (define what each app can do/access unlike SWIRLS's data flow-based policies). Finally, Knox cannot support simultaneous multi-context data access by the same app.

8 Related Work

Strict app sandboxing has been proposed in FlaskDroid [19] and Saint [70] that extend the existing permission policies by Android apps. Bluebox [16] provides a per-app data encryption mechanism, and corporate data access tracking. SWIRLS goes one step further and provides multiple contexts that are centrally monitored. Unlike Bluebox, SWIRLS does not rely on any network tool to detect data leakages, and tracks the data flow and enforces the policies within the system locally.

IFT-based solutions such as operating system-level frameworks, e.g., Asbestos [31], HiStar [96], TaintDroid [35], Flume [56], employ kernel-enforced mechanisms [31, 96], Dalvik machine-enabled techniques [35] or user-space engines [56] for IFT across system objects. CleanOS [86] introduces a security-enhanced garbage collector to protect sensitive data objects using encryption and data eviction. AppFence [51] suggests providing fake information when apps ask for sensitive user data. DroidScope [94] introduces offline VM introspection for IFT on Android emulators. Aquifer [68] and IpShield [21] prevent data leakages using LSM-based and Android sensor-level data protection, respectively.

Android permission system limitations or misuse is a well-studied problem [29, 77,

40, 41, 34, 51, 22]. SE-Android [81] deploys SE-Linux for Android at the kernel and middleware levels. Porscha [69] provides policy-based digital right management for smartphones without tracking the sensitive data flows. Trustdroid [18] provides a two-context work/personal domain isolation based on Tomoyo Linux. DeepDroid [91] enforce isolation policies at system server level. These MAC solutions rely on statically defined policies and do not support on-the-fly policy installation and enforcement. FlaskDroid [19] and ASM [49] provide kernel and user-space hooks for developers, however, the scope of the policy enforcement are limited by the hooks' points.

9 Discussion

Like most of existing practical IFT technology today, SWIRLS is unable to handle termination, timing, and implicit flows that can be used to circumvent data isolation.

Even coarse-grained isolation techniques like virtualization do not always provide protection against covert channel attacks by apps.

Rather than trying to improve on IFT, SWIRLS's goal is to enable BYOD through a more fine-grained and user-transparent data isolation model that gives data owners (not apps) control over data protection policies. Until IFT advances enable covert channel detection, we cope through a judicious choice of goals and threat model.

Additionally, the SWIRLS TCB contains the Android system (kernel, Dalvik, system services), and the SWIRLS server (Section 4). This TCB does not prevent a user from rooting the device and replacing the kernel to circumvent SWIRLS, no different from other isolation solutions available today. To prevent those attacks, hardware support such as TPM or TrustZone [5] are needed to ensure that the TCB has not been tampered with, no different from what some manufacturers do today, e.g., Samsung Knox on Galaxy S series phones [75]. We consider this problem outside the scope of this work.

10 Conclusion

SWIRLS enables BYOD through dynamic virtual micro security perimeters, i.e., capsules, to protect data and data owners rather than apps and services within a system.

SWIRLS keeps track of each capsule boundary across the system, and enforces relevant policies by deploying intra- and inter-process level mandatory access control. Our implemented fully-working SWIRLS prototype (>25K LOC) runs on Android `4.1.1_r6`, causes a reasonably low overhead on the system's overall throughput, and facilitates two appealing BYOD use-cases for data interactions between corporates and their employees.

Chapter 3

Value-Based Information Flow Tracking

1 Introduction

Modern mobile devices embed a wide range of sensors that enable new usages such as context awareness, activity recognition, and exercise tracking. Users have widely adopted these new mobile device capabilities as they do not require purchasing extra hardware but simply installing an application (app) that provides the new functionality. While these new sensors and applications empower users by providing useful features, these same sensors have also been exploited for malicious purposes. For example, previous research has shown that sensors such as accelerometer, gyroscope and ambient light sensors can be used as a keylogger mechanism [66, 82].

Mobile operating systems currently offer only rudimentary protection mechanisms to defend users against malicious inference attacks. The most popular mobile operating systems on the market, Google's Android and Apple's iOS, use runtime permission mechanisms to regulate application access to privacy-sensitive sensors, such as GPS or microphone. Other sensors such as the accelerometer do not require any permission at all. Permission mechanisms have been proven of limited efficiency: they provide coarse-grained permissions with no alternative but to comply with the permission request [76, 39]. As a result, it is nearly impossible for a user to grasp if any app computes a specific inference and if an app intentionally, or maliciously, leaks sensor values.

Information flow tracking (IFT) solutions have been proposed to identify applications that leak sensitive information. These solutions monitor data flows from a privacy-sensitive *source*, e.g. sensor readings, and determine if a flow of sensitive data reaches a *sink* which can be a network socket, a file, or a message shared with another app via inter-process communication (IPC). State of the art approaches for information flow

tracking analysis techniques can be categorized as follow: off-line static analysis (e.g., FlowDroid [10] or Droidsafe [46]) and runtime dynamic analysis (e.g., BayesDroid [87], TaintDroid [35] or Droidscope [94]). Both approaches have limitations. On one hand, static analysis suffers from high processing cost and can be easily bypassed by dynamic code loading. On the other hand, dynamic IFT solutions differ in the granularity they provide: for example, HiStar [96] labels operate on high-level system objects such as processes and files while TainDroid implements a variable-level IFT. This difference directly impacts the precision of the information flow detection. Moreover, dynamic analysis solutions tend to provide only minimal information in their leak reports. A typical alert currently includes the tainted data along with its corresponding source. It is then a challenging task to investigate qualitative aspects of a leak. Specifically, beyond the type of source reporting, a user might wonder how much information from a specific source a leak contains.

Dynamic IFT solutions provide "black-and-white" conclusions about whether a given app discloses sensitive data to unauthorized parties. The common flow provenance reporting adopted by IFT solutions fails to characterize a flow in several ways. Consider two different fitness sports apps, where one sends out accelerometer data readings constantly while in use, whereas the other app only reports monthly running distance averages. A typical dynamic data flow tracker would detect both flows since the network outputs are tainted by the source sensor data. However, a single value detected at a sink does not give any information about the type of computation it results from. Currently, taint tracking solutions do not differentiate a raw data leakage (e.g., raw accelerometer data) from an inference computed value (e.g. number of steps inferred from the raw accelerometer data). Static analysis solutions partially bridge this gap, but at a very high computing cost, in terms of both space and time. Such an extreme black-and-white treatment of sensitive data disclosures lead to unnecessarily pessimistic conclusions about legitimate apps. Consequently, users would discard reporting alerts due to their inaccurate reports.

A second limitation concerns the usability of current IFT solutions. Existing dynamic taint tracking solutions such as BayesDroid, TaintDroid or DroidScope require

system modifications. Generally, dynamic information flow solutions require substantial system modifications and require the user to install a custom modified version of Android and/or require root privileges. Such requirements have prevented the adoption of information flow tracking technologies by common users who are incapable or unwilling to replace their factory images of Android with a custom system in order to check if an application is disclosing their personal data. As a result, the investigation of privacy leakage by applications has been the work of a limited group of researchers that do not have the resources to cover the gigantic amount of applications available on smartphone platforms. Also, currently available solutions such as TaintDroid [35] are implemented by modifying the Dalvik virtual machine. Unfortunately, these solutions are not compatible with new versions of Android since Google has shifted the Android application execution environment from the Dalvik virtual machine to the new Android runtime (ART) as of Android 5.0. Recent work such as ARTist [12] proposes an ART-based taint tracking but still requires system modifications.

We introduce METRON, a new practical information flow tracking solution and data disclosure analysis framework for numerical values such as sensor measurements. To address fundamental issues in current taint tracking solutions, METRON focuses on three main aspects. First, METRON introduces a new value-based information flow tracking system. Second, METRON tracks tainted data operation history and adds this information to the sink reporting. Third, METRON's design allows for deployment on top of unmodified commodity versions of Android without requiring root access. Hence METRON can be used by any user to check the privacy leakage of applications running on her phone.

While the majority of previous IFT solutions [35, 94, 96] use a per-object shadow memory (*taint*) to identify data flows, METRON introduces a new approach for tracking sensitive data based on the sensitive data values. This chapter provides an evaluation of this value-based IFT approach and shows how it achieves comparable performance and precision to previous solutions while incurring less computation and memory overhead, and having fewer deployment constraints. In addition to this flow detection mechanism,

METRON records each taint value arithmetic operation history. This enables a characterization of data flows beyond a white and black assessment (i.e., whether a data segment is tainted).

We evaluate this approach through a prototype of METRON implemented on top of Android 5.0. The prototype runs as a third-party application that acts as a sandbox layer between the unmodified Android system and the monitored application. METRON evaluation is carried on unmodified Android systems running real world applications installed directly from Google's Play Store. In addition, METRON's design was ported back to Android 4.3 in order to compare its accuracy against the state of art legacy solutions such as TaintDroid [35], and BayesDroid [87]. Compared to TaintDroid, METRON improves the detection accuracy significantly and reports fewer false positives (2 vs 17 on DroidBench) mainly because of variable sensitivity considerations (i.e., a variable tainted and reused with an untainted value stays tainted). Compared to BayesDroid, METRON reports higher accuracy in terms of the number of true positives while being able to handle numerical values whereas BayesDroid can only track strings.

While the Android framework provides a great platform to evaluate such a solution considering the previous research effort, we strongly believe that this approach would benefit even more domain-specific applications such as Internet of Things devices that manipulate mostly sensor data and numerical values.

The contributions of this chapter are as follows:

- We designed a novel lightweight information flow analysis algorithm for numerical values that does not require an exhaustive instruction instrumentation.
- We implemented a prototype of METRON and evaluated its performance and accuracy versus other solutions. This evaluation was conducted using benchmarks and real-world popular applications' execution traces that manipulated sensors values.
- We improved on current IFT solutions capabilities by providing insightful information about leaks that go beyond a true/false detection mechanism through the collection of tainted data operations history as an inference detection mechanism.

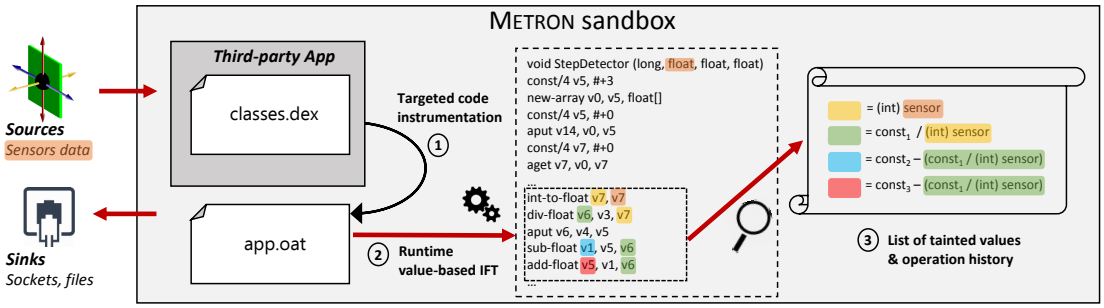


Figure 3.1: Overview of the system: selective instrumentation of tracee application instructions and extended flow reporting

The rest of this chapter is organized as follows: Section 2 provides a high-level overview of METRON. Section 3 presents the threat model for METRON's current implementation. Section 4 and Section 5 present the design of METRON's value-tracking approach and detail its implementation. Section 6 describes our experiments and evaluation results. Section 7 discusses the limitations and future directions of our work. Section 8 summarizes the related work. Finally, Section 9 concludes the chapter.

2 Metron Overview

Table 3.1: Number of floating point operations encountered at execution time in the Cfree pedometer app

# DEX op executed	25,080,000
# fp op executed	106,514 (0.42%)
# numerical op executed	1,891,834 (7.5%)

METRON is a dynamic information flow tracking (DIFT) framework for numerical values. Figure 3.1 presents the overview of the framework. METRON sources can be any system component that generates numerical values, such as phone sensors. Contrary to existing techniques [35, 94] that use a per object shadow memory to track information flows, METRON employs a novel data flow tracking technique based on examining the operands and return values of numerical operations. METRON keeps track of the result of numerical computations that involve tainted values which are either raw values from a source or computed values that are functions of other previously tainted values. METRON operates by instrumenting numerical instructions in order to track operand

values and record return values as a flow tracking mechanism. When data reaches a sink, METRON detects the information flows involving tainted values by comparing string representations of numerical values against the set of previously encountered tainted values. The inspection of numerical operations operands and return values is achieved by running the app within a virtualized environment (i.e., a sandbox) and by instrumenting the application code using an modified version of the `dex2oat` compiler. This sandbox and modified compiler are implemented and run as a third-party Android application.

Most Android applications execute a small ratio of numerical instructions compared to the overall number of instructions in an app. Table 3.1 shows the percentage of numerical and, more specifically, floating point operations in a popular pedometer application. Such an application, which constantly computes inferences from sensor data, has a ratio of numerical instructions to the total number of instructions performed $< 10\%$ (and $< 1\%$ for floating point operations). This observation motivates the design choice of selectively monitoring numerical operations. Instead of instrumenting all operations to detect information flows, METRON only instruments the subset of instructions which handle numerical values.

This selective instruction coverage implies a tradeoff between METRON's accuracy versus traditional approaches for taint-tracking (e.g. taint stored in shadow memory). We investigate this tradeoff by analyzing the limitations in terms of flow coverage and correctness of the flows detected. Intuitively, limiting the scope of operations tracked to numerical operations limits of the detection precision of the system. In practice, our experimental evaluation of METRON reveals that this limitation does not heavily affect the solution flow detection rate in comparison to other solutions. Also, employing the values themselves as a propagation mechanism can lead to false positives: an identical numerical value can be read multiple times from a sensor or can result from two different sequences of operations, from two different data flows. As discussed in Section 4, such false positives are monitored and mitigated through METRON's design. In practice, we did not observe many false positives resulting from value collisions. Section 6 discusses this point further.

METRON provides extra information to the user regarding the nature of the flow that reaches the sink (Figure 3.1). METRON keeps track of the computation history of each tainted numerical value. This extra information provides a basis for further analysis such as deriving a risk analysis based on the history of operations used to compute the leaked value. While METRON provides this operation history tool for further analysis, this research does not claim to evaluate the risk of observed data leaks. In fact, the determination of a risk is closely related to specific use cases and to users privacy expectations. Accordingly, METRON's contribution is limited to the framework to investigate data leakages. The risk assessment of a flow reaching a sink is left for future research.

The user is required to install two apps to use this solution: the METRON app and the application to be investigated. The METRON app tracks data leakage from any other third-party apps, while not requiring any changes to the system or to the investigated app. The target application is started through METRON. METRON transparently uses dynamic code loading, compiler instrumentation and app virtualization techniques to monitor the flows of sensitive values while the target app is running. Our implementation is discussed in Section 5.

Our investigation of this novel IFT mechanism focuses on accelerometer-based applications. Many accelerometer inferences have been proposed in previous work for applications as diverse as detecting stress [43], remotely capturing keyboard input [64] or providing fitness tracking information. Such a diversity of computations provide a great opportunity to evaluate our solution.

3 Threat Model

METRON threat model assumes that apps have full access to the device sensors and leverage this access to achieve their purpose. This solution considers data leaks caused by legitimate (as opposed to malicious) applications. We assume that applications examined by METRON do not try to leverage any known or unknown vulnerability in the underlying execution framework (kernel, system services). While the implementation

presented in this chapter follows this assumption, the design proposed could have a relaxed threat model that only assumes that the hardware is trusted by leveraging trusted hardware commodities such as ARM TrustZone to host the flow decision logic and tainted values.

We assume that applications do not use any type of covert channel attack. Also, we assume that no external actors try to defeat the analysis techniques presented by tampering with the sensor environment. METRON does not detect information leaks through side channels and assumes that there is no intentional tampering with the phone environment to defeat the flow detection. Implicit data flows are also not covered by METRON.

4 Value-Based Information Flow Tracking

This section presents the design challenges that were addressed while designing METRON value-based information flow tracking system.

4.1 Sources and Sinks

METRON information flow tracking focuses on numerical value data flows. The information sources are numerical value generators such as the accelerometer, the GPS, and the heart-rate sensor. Generally, sensors that generate floating point values are METRON targets of choice as data sources. This selection of sources fits well for smartphone and IoT platforms: [28] enumerates the motion sensors supported by the Android OS and their unit of measure. Most of these sensor readings are represented as floating point values in Android and Android Things (formerly Brillo).

METRON uses classic detection interfaces as information flow sinks. Possible sinks include network sockets, IPC messages, and files. At the sinks, METRON assumes the data is encoded as cleartext ASCII byte arrays. Data leaks are detected by comparing all numerical value representations in the string to the list of tainted numerical values maintained by METRON. This detection relies on a regular expression that matches any integer or floating point representation.

Listing 3.1 shows an example of a file written by a pedometer application, installed from the Google Play Store, that utilizes the accelerometer to compute the number of steps the user makes. On the file write operation, the string given as an argument is detected at the sink and marked as containing sensor data. Not all integer or floating point values carry sensitive information in this string. In our example, the XML version number, 1.0, and the 8 of `utf-8` are not tainted. However, the remaining numerical values in this file are tainted values that derive from a computation from accelerometer readings. In this case, the flow detection is triggered by the value *0.010439022*. The value 25 actually results from an implicit flow and is not detected. Such detection results are similar to other dynamic IFT solutions: they indicate the origin of the data, the source and the sink the tainted data was detected at.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <int name="lapsteps" value="25" />
    <float name="distance" value="0.010439022" />
    <float name="lapdistance" value="0.010439022" />
    <long name="lapsteptime" value="11670" />
    <long name="steptime" value="11670" />
    <float name="calories" value="0.85382223" />
    <float name="lapcalories" value="0.85382223" />
    <int name="lapnumber" value="1" />
    <int name="steps" value="25" />
</map>
```

Listing 3.1: Example of a flow detected (App: Accupedo)

4.2 Tainted Data Computation History

METRON improves on existing IFT solutions by recording the operation history each tainted value is derived from. As presented in Listing 3.2 each source reading is represented by a symbol (e.g., S0 the for accelerometer on the X-axis) and an index to indicate the order the value was read from the sensor. Each raw or computed tainted value possess a computation history string that is eventually displayed at the sink if the value is leaked.

```
(((((0.0909091*S0_114)+(0.909091*((0.0909091*S0_138)+(0.909091*((0.0909091*S0_36)
     +(0.909091*((0.0909091*S0_18)+(0.909091*((0.0909091*S0_120)+(0.909091*((0.0909091*S0_114)
     +(0.909091*((0.0909091*S0_12)+(0.909091*((0.0909091*S0_102)+(0.909091*((0.0909091*S0_0)
     +(0.909091*((0.0909091*S0_30)+(0.909091*((0.0909091*S0_0)+(0.909091*((0.0909091*S0_6)
     +(0.909091*((0.0909091*S0_36)+(0.909091*((0.0909091*S0_0)+(0.909091*((0.0909091*S0_60)
     +(0.909091*((0.0909091*S0_54)+(0.909091*((0.0909091*S0_12)+(0.909091*((0.0909091*S0_0)
     +(0.909091*((0.0909091*S0_36)+(0.909091*((0.0909091*S0_30)+(0.909091*((0.0909091*S0_24)
     +(0.909091*((0.0909091*S0_18)+(0.909091*((0.0909091*S0_12)+(0.909091*((0.0909091*S0_6)
     +(0.909091*((0.0909091*S0_0)+-0.0579258))))))))))))))))))))))))))))))))))))))))))))))))
)))
```

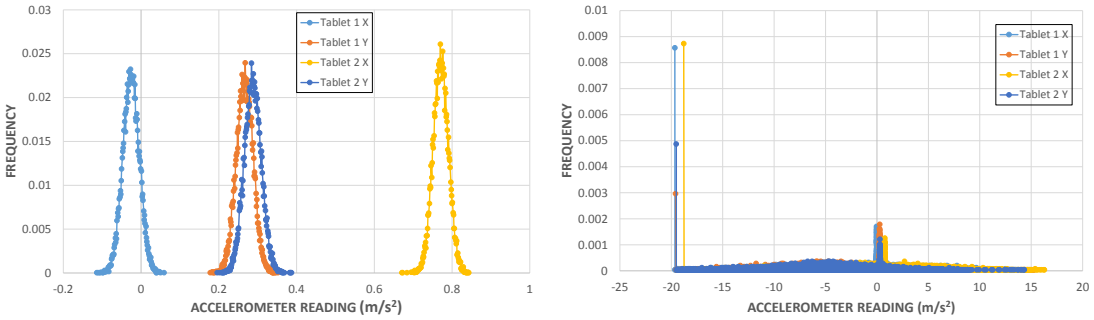
Listing 3.2: Example of a flow history at the sink. The S0 prefix corresponds to the accelerometer on the X-axis and the suffixes correspond to the order the values were read from the sensor. (App: Accupedo)

4.3 Design Challenges

Contrary to metadata-based information flow tracking solutions where system objects such as variables, process and files are tainted using shadow variables or file metadata to indicate the belonging or the provenance of a flow, METRON relies on the comparison between the numerical instruction operands and return values as a primitive to track a data flow. If any of the operands belong to a list of tainted values maintained by METRON, then the return value of the instruction is also added to the list of tainted values. The list of tainted values is bootstrapped by the raw values obtained from the source.

We discuss some design choices in METRON and their limitations. Using values as a primitive to track flows requires investigating a few critical aspects: (1) the requirements for unique values to achieve an accurate flow detection (or flow computation history reconstruction), (2) the design of a storage method for values that belong to a flow, and (3) a value lookup mechanism. The rest of this section describes our approach for each of these aspects. Section 6 complements our discussions with actual measurements.

Numerical Values Unicity Requirements. Two levels of operation of METRON's information flow tracking solution should be distinguished: (1) the ability to differentiate a tainted data flow from a non-tainted data flow and (2) the ability to identify a flow along with its computation history. These two levels of operations have different requirements regarding the values that are processed. Tainted value collisions (i.e., identical values read or computed from a sensor at different times) have no incidence



(a) Distribution of accelerometer readings for static tablets

(b) Distribution of accelerometer readings for randomly moving tablets

Figure 3.2: Accelerometer reading collision frequencies for two tablets attached together on the X and Y axes over a 120s time period.

on the detection of a flow. However, an untainted computed value could coincidentally equal to a tainted value. This case need to be handled as it would lead to a false positive (**Requirement A**). This first requirement also holds when trying to reconstruct a flow computation history. Moreover, the distinction between two identical tainted values that result from different computations (e.g., a raw tainted value vs. a computed tainted value) need to be made (**Requirement B**) as they carry different computation histories. Also, the unicity of the values read from the sensor matters as they correspond to the order the values were read at (**Requirement C**).

Requirements A and B are strong requirements since the correctness of the solution depends on them. However, requirement C only matters if the analysis of a flow computation history requires information about the order of the sensor readings. For example, the computation of a mean value of a set of sensor readings would not require knowledge about the order of acquisition of the values. On the other hand, this order might be useful to further investigate a raw data leak.

To evaluate how unique values generated from a sensor are, we consider the distribution of accelerometer readings generated from two different tablets. We first sampled them at 40Hz per axis over 120s time periods first by keeping them still on a table then by moving them together randomly over a same time period. Figure 3.2a and Figure 3.2b show the distributions of the accelerometer readings for two identical devices in both cases. For the static devices, the measurements revolve around 0 for x and y and

Table 3.2: Entropy, ratio of unique values, minimum, maximum and mean time intervals in seconds over which a static and moving device acquired unique accelerometer readings over 120s time frame for two identical devices.

Axis	Device	Static					Moving				
		Ent.	**% uniq.**	**Min**	**Max**	**Avg**	**Ent.**	**% uniq.**	**Min**	**Max**	**Avg**
X-axis	Tablet 1	4.29	1.06	0	0.153	0.048	9.21	53.90	0	2.389	0.140
	Tablet 2	4.22	0.51	0	0.139	0.088	9.27	54.15	0	1.746	0.352
Y-axis	Tablet 1	4.31	0.56	0	0.136	0.060	9.10	47.77	0	1.440	0.093
	Tablet 2	4.32	0.56	0	0.141	0.008	9.16	48.32	0	1.632	0.294
Z-axis	Tablet 1	5.07	1.02	0	0.241	0.037	6.84	45.60	0	0.821	0.046
	Tablet 2	5.05	1.06	0	0.208	0.089	6.94	46.49	0	1.090	0.004

Maximum entropy (i.e. all distinct values) for the collected sample size: 10.11

G for z (not shown in the graphs). The readings collision frequency generate Gaussian distributions for static devices and more random distributions for the moving devices. We estimated the entropy of these sets of values by empirically computing Shannon's entropy approximation using the approach mentioned in [17]:

$$H_s = -\sum_{i=1}^{M} p_i \, ln(p_i) = -\sum_{i=1}^{M} \frac{n_i}{N} \, ln(\frac{n_i}{N})$$

where M is the set of unique values in the set of all values read N, p_i is the probability of reading the value i from this set, n_i is the number of occurrences of value i.

Table 3.2 presents our per-axis and per-device results. While there is a significant increase in entropy for the moving devices the ratio of unique values was 50% maximum over 120s. Table 3.2 also presents the minimum, maximum and mean time intervals over which we acquired unique values from a sensor. The average time intervals over which we can acquire unique values are usually less than a second. While it is possible to have unique values over a small period of time, a solution to consistently achieve the unicity of all values read from the sensor (Requirement C) is required.

A simple solution consists of adding some negligible perturbation (much smaller than the sensors' typical noise levels) to colliding values. By adding a very small variation to the sensor values that stays within the limit of the bias of the sensor, we were able to achieve a full set of unique values while minimally perturbing the input. For the

test above, we used two Nexus 7 tablets both equipped with a MPL accelerometer from Invensense. According to the results from the SensMark benchmark [78], the resolution achieved by this sensor is $0.039m/s^2$, with a detection range between $\pm 19.6133m/s^2$. On Android, accelerometer values are stored as floating point values that can achieve up to nine significant decimal digits precision. This leaves sufficient room for our sub-noise-level perturbation that eventually has no impact on the precision of the reading and computations afterward, since the conventional measurement noise should already be taken care of by data-consuming apps on the phone.

Finally, Requirement A can be monitored by checking that no result from untainted values matches a tainted value. Similarly, Requirement B requires looking up known tainted values before inserting a new value. METRON raises an alert to indicate any of these cases. In practice, while running METRON alongside with TaintDroid, we did not observe false positives due to either of these two cases.

Storage of Tainted Values. Despite the recent progress made by hardware constructors, smartphones are still embedded devices with limited resources. Android operating system designers constrain the use of memory under a certain threshold by using mechanisms such as Android's *lowmemorykiller*. METRON's design for value storage adapts to this constraint by using a fixed per-app memory space to store tainted values. METRON employs a ring buffer where only the most recent tainted values are kept and the oldest ones are reused. Section 6.1 goes into the detail of this choice. We also store the history of the computation applied to a tainted value along with the value. This history consists of tainted values each represented as an arithmetic function of the original sensor readings based on the computations performed by the app up to this point. This feature requires a larger memory space as the computation history depends on the underlying app algorithms.

Tainted Values Lookup. METRON uses a lookup mechanism on the ring buffer in order to determine if a value is a part of a flow. Based on our observations, most of the lookups match values at the beginning of the buffer, which is logical since a sequence of

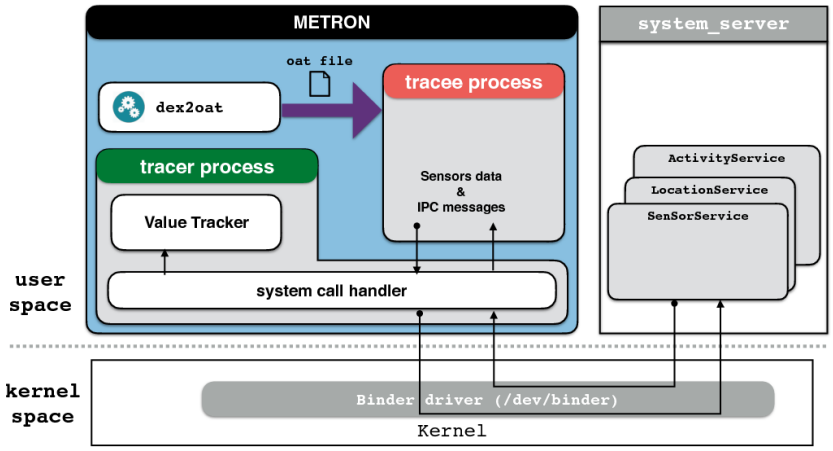


Figure 3.3: Overview of METRON's app components

operations in the code will often reuse the most recent computation results. Accordingly, METRON uses a hybrid lookup mechanism that first searches the most recently inserted values in the ring buffer and then uses a full binary tree search if the value was not found. One important point to note, depending on either METRON is run to only track flows or to additionally track flow computation histories, respectively one or multiple lookups in the ring buffer are required. In order to detect a flow, a positive lookup on any of the parameters that are given to a numerical operation is sufficient to detect a flow. However, if METRON is employed to build the history of the flow, it is imperative to look up all operands of the operation in order to build the precise operation history linked to the flow. The taint propagation corresponds to the search for an exact match between the values given as a parameter of an instruction and the values stored in the ring buffer. At the sink, however, we look for an approximate match between the value written to the sink and the values known in the ring buffer since developers sometimes limit the number of decimals printed through printing functions modifiers.

5 Implementation

METRON is implemented as a *userspace* third-party app compatible with the modern versions of Android (5, 6 and 7). We carried out our evaluation experiments on Android 5.0.1. Since version 5.0, Android uses by default a new runtime mechanism, named Android Runtime (ART), which uses ahead-of-time compilation to compile DEX

applications bytecode into an ELF binary file (more specifically, OAT file format) at application installation time. This compilation takes place on the device using the `dex2oat` compiler. While switching from Dalvik to ART breaks legacy taint tracking solutions (e.g., TaintDroid), METRON is compatible with the new runtime environment. METRON can be used on commodity Android devices without modifying the operating system nor the application to be examined.

To investigate data leaks in an installed third-party app, the user has to start the target application through METRON's app. METRON relies on two mechanisms to perform its analysis: (1) an app sandboxing mechanism and (2) a recompilation of the app DEX bytecode using METRON's modified `dex2oat` compiler shipped as a binary executable, inside the METRON app.

The app sandboxing mechanism enables the inspection of data flowing in and out the target app. This sandboxing is achieved through system call interception. The modified compiler enables tainted value tracking by instrumenting DEX numerical instructions present in the target app DEX code. When an app is run for the first time within METRON, the modified compiler generates an instrumented OAT file from the app DEX bytecode. METRON does not replace the target app OAT file in the system with our modified version (which is impossible because METRON does not have root access), instead when an app is selected to run within METRON, we rely on system call interception to redirect the `open` system call to the path of the modified OAT file generated inside METRON's app.

Figure 3.3 shows METRON's architecture. At runtime, METRON manages two processes. The first one is the *tracee* process within which the modified binary file of the target app is loaded and executed. The other one is the *tracer* process that intercepts all system calls, including sensor data delivery, files and network accesses, and IPC messages between the tracee process which runs the target app and the Android OS. Data leak detection also relies on system call interception: when reaching a sink METRON tracer compares the values being sent out the tracee process against the list of tainted values maintained by the value tracker.

The next subsections 5.1 and 5.2 respectively detail the design and implementation

of the application sandboxing and the instrumented compiler.

5.1 Application Sandboxing

The core component of METRON is an application sandboxing process that is used to execute the code of other third-party applications. Techniques to implement application sandboxes have been demonstrated in [15, 11]. Both approaches use dynamic code loading and system call interposition to implement the sandboxing mechanism. Our implementation extends these approaches and adds information flow tracking through the METRON app.

First, we use Android dynamic code loading APIs to load the code of the other application. Second, within the main process of METRON we set up an environment for intercepting system calls (including the `ioctl` system call used to perform Binder's IPC transactions). Lastly, we start the main activity of the target app while using system call interposition to inspect and modify the content of the system calls and Binder transactions payloads.

Target App Code Loading. Android provides well-documented APIs for dynamic code loading. The `DexClassLoader` class loads classes from a given JAR or APK file. In addition, Android provides the `createPackageContext` to create the context object of a given installed application. The returned context object includes the application resources from its APK file and optionally, can include the application code as well.

```
Context target_context = getApplicationContext().createPackageContext(
    targetPackageName,Context.CONTEXT_IGNORE_SECURITY | Context.
    CONTEXT_INCLUDE_CODE);
ClassLoader loader = target_context.getClassLoader();
```

The security implications of dynamic code loading have been discussed in [38].

System Call Interception. Before starting the code loaded from the target application, METRON enables system call interception in the tracer process. This is done using

the ptrace system call. First, METRON main app process uses the fork system call to clone itself into another process. The new child process becomes the tracer process. After making the fork system call, the parent process, the tracee, will allow the tracer to use ptrace to control it via by the following code:

```
prctl(PR_SET_DUMPABLE, 1, 0, 0, 0);
```

Listing 3.3: prctl system call

The tracer will then make the following system call to attach its parent:

```
ptrace(PTRACE_ATTACH, parent_pid, 0, 0);
```

Listing 3.4: ptrace system call to attach a process

As a result, the main process becomes a tracee controlled by the tracer process. Consequently, the tracer will get interrupted for every system call made by the tracee. When interrupted, the tracer is able to read and modify the registers and memory content of its tracee.

Target Application Runtime Monitoring. With the system call interposition ready, the tracee starts the main activity of the target application by using the ClassLoader we created from the target application context to load the main activity class. The loaded classes are wrapped with an Intent wrapper and ask the ActivityManager frameworks service to start it by using StartActivity function call. While the normal behavior of Android is to disallow applications to start the code loaded from other application packages in the same process, we make use of the system call interception to modify the payload of the Binder transactions made by the StartActivity framework method call. Namely, we use the tracer process to modify the content of both the START_ACTIVITY_TRANSACTION, and SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION taking place between the METRON application process (the tracee) and the Android framework ActivityManagerService process. More details about the exact patching procedure can be found in [15].

Eventually, while the target app is running, the tracer process performs the following tasks:

a) Virtualizing the resources of the target app: In order to keep the target app running within the context of our tracee process, the tracer has to virtualize the private resources of the target app such as the application components (e.g. Activities, Services) and the private data directory (`/data/data/[application_package_name]`). Files accessed under the private data directory of the target app package are redirected to the private data directory of METRON by intercepting the file-related system calls. When the target app attempts to start another activity from its own package, the tracer intercepts the Binder transactions to replace the package name of the activity to be started with the package name of the METRON app. When the `ActivityManagerService` responds by providing the `ActivityInfo` of the activity to be started, the tracer replaces it by an instance of the `ActivityInfo` obtained from the target app context we obtained by dynamic code loading.

b) Intercepting sensor values delivered to the app: The tracer process intercepts IPC messages between the tracee process, which runs the target app, and the sensor data sources within Android (e.g. `SensorManagerService`). For example, to intercept accelerometer and gyroscope values the app receives, we intercept the `GET_SENSOR_CHANNEL` exchanged between the application and the framework `SensorManagerService` which contains a descriptor for a network socket used to deliver sensor values to the application. The network socket (`recvfrom`) system call is intercepted to inspect the actual values delivered via the socket descriptor we have found. Sensor values are added to a list of sensitive values maintained by METRON.

c) Intercepting file and network-related system calls: The tracer will intercept file and network related system-calls to inspect the data written to files and network sockets for tainted values. Values written to files or network sockets are checked against the list of sensitive values maintained by METRON.

5.2 Taint Tracking via Numerical Operations Interception

The sandbox system call interception allows the tracer to inspect values read from sources (e.g., sensor readings from the accelerometer are received through a network socket `recvfrom` system call) and values sent to sinks (which are leaked by either

```
float f1 = 4.05f;
float f2 = 5.04f;
float f3 = f1+f2;
float f4 = f2-f3;
```

Listing (3.5) Sequence of floating point operations in an Android app

```
1: void com.example.floattrack.
    MainActivity$1.onClick(android.view.
    View) (dex_method_idx=18789)
DEX CODE:
 0x0000: const v0, #+1082235290
 0x0003: const v1, #+1084311470
 0x0006: add-float v2, v0, v1
 0x0008: sub-float v3, v1, v2
```

Listing (3.6) Generated DEX code extracted with `oatdump`

```
   rl_src1 = LoadValue(rl_src1, kFPReg);
   rl_src2 = LoadValue(rl_src2, kFPReg);
   rl_result = EvalLoc(rl_dest, kFPReg,
       true);

+  NewLIR1(kThumbPush, rs_r0.GetReg());
+  NewLIR1(kThumbPush, rs_r7.GetReg());
+  LoadConstant(rs_r7, 382);
+  NewLIR1(kThumbSwi, 0);
+  NewLIR1(kThumbPop, rs_r7.GetReg());
+  NewLIR1(kThumbPop, rs_r0.GetReg());
+
   NewLIR3(op, rl_result.reg.GetReg(),
       rl_src1.reg.GetReg(), rl_src2.reg.
       GetReg());
   StoreValue(rl_dest, rl_result);
```

Listing (3.7) Approach 1: `dex2oat` compiler numerical instruction instrumentation

```
ed9f9ab5        vldr.f32 s18, [pc, #724]
     ; 0x4081999a
ed9f8ab3        vldr.f32 s16, [pc, #716]
     ; 0x4081999a40a147ae
+  b480         push r7
+  b487         push r0
+  f44f77bf     mov.w   r7, #382
+  df00         swi 0
+  bc87         pop r0
+  bc80         pop r7
ee798a08        vadd.f32 s17, s18, s16
+  b480         push r7
+  b487         push r0
+  f44f77bf     mov.w   r7, #382
+  df00         swi 0
+  bc87         pop r0
+  bc80         pop r7
ee789a68        vsub.f32 s19, s16, s17
```

Listing (3.8) Approach 1: Generated assembly code by Metron's `dex2oat`

```
    case Instruction::ADD_FLOAT:
    case Instruction::SUB_FLOAT:
    case Instruction::MUL_FLOAT:
    case Instruction::DIV_FLOAT:
    case Instruction::REM_FLOAT:
    case Instruction::ADD_FLOAT_2ADDR:
    case Instruction::SUB_FLOAT_2ADDR:
    case Instruction::MUL_FLOAT_2ADDR:
    case Instruction::DIV_FLOAT_2ADDR:
    case Instruction::REM_FLOAT_2ADDR:
+     GenInvoke(metron_callinfo);
      GenArithOpFloat(opcode, rl_dest,
          rl_src[0], rl_src[1]);
      break;
```

Listing (3.9) Approach 2: `dex2oat` numerical instruction instrumentation

```
ed9f9ab5        vldr.f32 s18, [pc, #724]
     ; 0x4081999a
ed9f8ab3        vldr.f32 s16, [pc, #716]
     ; 0x4081999a40a147ae
+ 1c38         mov     r0, r7
+ 68c0         ldr     r0, [r0, #12]
+ 6980         ldr     r0, [r0, #24]
+ f8d0e028     ldr.w   lr, [r0, #40]
+ 47f0         blx     lr
ee798a08          vadd.f32 s17, s18, s16
+ 1c38         mov     r0, r7
+ 68c0         ldr     r0, [r0, #12]
+ 6980         ldr     r0, [r0, #24]
+ f8d0e028     ldr.w   lr, [r0, #40]
+ 47f0         blx     lr
ee789a68          vsub.f32 s19, s16, s17
```

Listing (3.10) Approach 2: Generated assembly code by Metron's `dex2oat`

Figure 3.4: Instrumentation of the app code by Metron's compiler and corresponding implementation of the compiler for a function call value interception.

network sockets, written to file, or passed through Binder IPC to another application). However, system call interception does not provide information about the computations applied to the tainted values after being read from the source and before being sent to the sink. Without this information, our information tracking solution would only be able to identify direct leaks from a source and would not be able to detect leaks that follow computed inferences from the sensitive values. In order to overcome this hurdle, we rely on compiler instrumentation by shipping a modified version of the `dex2oat` compiler within the METRON application.

Starting from Android 5.0, the ART runtime environment has been used as a replacement for Dalvik. One of the main features of this new runtime environment is the *Ahead-of-Time* compilation that transforms the DEX bytecode that is embedded in an android application to optimized native code. This ahead-of-time compilation is a one-time operation performed during the app installation. The compilation process is performed using the `dex2oat` utility that acts a compiler for DEX bytecode. The resulting binary is an ELF executable with the application DEX code embedded in it for debugging purpose. The dex2oat compiler has different compilation modes. The default *quick* compilation mode uses Android's own compilation backend, as opposed to the *portable* compilation mode that relies on LLVM. The quick compilation takes place in two phases. The first phase transforms each sequence of DEX instructions into a list of instructions corresponding to the opcodes, referred to as the *MIR* representation. The second phase transforms the MIR representation to a platform specific representation, referred to as *LIR*. Both steps include optimization phases such as garbage collection optimizations and improved register mapping among others. Finally, the compiler generates platform-specific native code from the LIR representation.

During the first target app startup, METRON runs the modified compiler on the application's `classes.dex` file (which is accessible under the `/data/code/[package_name]` directory) to generate a modified OAT file. This compilation step attaches a METRON library DEX file to the target app DEX file that contains the numerical instruction hook functions in charge of keeping track of the data flows.

The modified `dex2oat` compiler instruments the DEX numerical operation instructions (add-*, sub-*, mul-*, div-*, etc) while converting them to native code. We describe below the two injection approaches that we evaluated to keep track of the tainted numerical values.

Approach 1: System Call Injection. Our first approach relies on system call interception. The compiler adds an arbitrary selected unused system call number, such as 382, right before any floating point operation in the code. Right before this syscall, the register 7 and 0 (that respectively contain the system call number and the return of the sys-call) are saved. The system call number injected is unknown to the kernel and thus no operation is performed by the kernel. However, it is intercepted by METRON sandbox as a marker that a floating point instruction is being executed. The tracer then inspects the registers of the target application process (i.e., tracee) and looks up both the operands and return value of the numerical operation. Finally, the syscall returns and the application execution is resumed by restoring the original values of the registers $R7$ and $R0$.

The placement of this system call before the operation is required in certain cases: instructions such as `add,sub,div,mul`-2addr use the first register to both provide an operand and store the return value. In that case, the return value is computed twice.

Listing 3.5 shows an example of Java code that manipulates numerical values. The equivalent DEX bytecode is given in Listing 3.6. The native assembly code generated via compiling this DEX bytecode by our modified compiler is shown in Listing 3.10. Instructions injected due to our compiler extensions are marked with '+' symbols.

In practice, this approach is quite rudimentary and globally leads to poor performance (cf. Section 6).

Approach 2: Function Call Injection. A more efficient approach to intercept numerical operation in the target program can be achieved by injecting a function call before floating point operations rather than triggering a system call. A function call injection prevents the tracee process interruption and the context switching overhead.

Instead, the floating point operation interception code is run inside the tracee process itself.

METRON's implementation for the function interception relies on a modified ART quick compiler. The modified compiler (1) identifies the function code to inject from the METRON interception library compiled along with the target app, and (2) injects a call to the value interception function on the target app right before every numerical operation seen on the DEX code. To include the code METRON uses to intercept function calls, we use `dex2oat`'s multi-file compilation support (`--dex-file` option) to combine the app's DEX file, obtained through the APK, with METRON's interception library.

Android leverages hardware functionalities, when available, to speed up floating point operations. In particular, Android makes use of the ARM Floating Point architecture (VFP) to handle floating point operations. The VFP architecture uses a set of registers of either 32 bits or 64 bits that are dedicated to floating point operations. We used the capstone [72] library to disassemble the floating point instruction to be executed next. We record the operation to be executed as well as the registers involved. By looking up the VFP registers that correspond to the operands we find the corresponding numerical values and look them up in the ring buffer.

When the values are intercepted, they are stored in the shared memory zone between the target application and the tracer process. Eventually, when the target application reaches a sink, which always happens through a system call, the tracer layer verifies the data passed as a parameter against the values in the ring buffer.

Listing 3.5 shows an example of Java code that manipulates numerical values. The equivalent DEX bytecode is given in Listing 3.6. The native assembly code generated via compiling this DEX bytecode by our modified compiler is shown in Listing 3.8. Instructions injected due to our compiler extensions are marked with '+' symbols.

Listing 3.7 presents a part of the modifications we made to the `dex2oat` compiler source code. Specifically, METRON adds a call to METRON's library interception function before numerical operations using the compiler `GenInvoke()` method.

Reference: Dalvik-Based Implementation

Lastly, in order to compare the precision of this solution versus existing solutions, we have also ported back METRON design to Android version `4.3_r1`, modified with TaintDroid patches, to compare METRON performance with previous implementations in Section 6.

6 Evaluation

Sections 6.1 and 6.2 detail the performance overhead and flow tracking accuracy results respectively.

6.1 Performance Overhead

The overhead added by METRON can be attributed to the following factors: (1) The overhead due to running the application inside the METRON sandbox instead of running it directly on top of the Android OS. (2) The overhead due to the additional instructions injected by the modified `dex2oat` compiler. (3) The overhead due to looking up the values of numerical operations operands into the ring-buffer to perform taint propagation. This section provides our evaluation of these three factors.

Sandbox. Running the app within a sandbox allows us to dynamically examine the app behavior while running without having to modify the operating system or the app (APK) package. This advantage comes at the cost of runtime penalty due to the interception and modification of the system calls and Binder transactions. We measure the virtualization overhead on popular real-world apps downloaded from the Google Play Store. We run each app in two modes: running natively on top of the OS and running within the sandbox. For each mode, we conduct five experiments and report the median time it takes to start the app's main activity. More precisely, we measured the time interval between the framework's `StartActivity` method invocation (app startup) and the end of the `OnCreate` method execution when the activity is started.

The result in Table 3.3 shows that the first two apps were launched around 36%

Table 3.3: App Launch Overhead (Without Compilation)

Application	**Native Exec.**	**Sandboxed Exec.**
SensorBox	$455\,ms$	$620\,ms$ (136%)
Linpack Mobile	$813\,ms$	$1119\,ms$ (137%)
Free Pedometer	$1442\,ms$	$8596\,ms$ (590%)

slower when they were started within the sandbox. The Free Pedometer application experiences a more severe slowdown during the launch because it performs more operations to load and initialize ads and application analytics libraries during its execution. Also, it performs multiple reads/writes to a private SQLite database.

Numerical Operations Instrumentation. The additional system calls or function calls that are injected by the modified `dex2oat` compiler in order to intercept the operands and return values of floating point operations also add a runtime overhead. To measure this overhead, we use the Mobile Linpack benchmark app, which is a CPU-intensive floating point benchmark. We compare the scores while running (1) natively on top of OS. (2) within sandbox with system call interception. (3) within sandbox with function call injection. The results are shown below in Table 3.4.

Table 3.4: Numerical operations benchmarks

Benchmark	**% Num. Op**	**Native Execution**	**Metron system call approach**	**Metron function injection**
Linpack Mobile	29.35	4.6 Mflops	0.01 Mflops	2.07 Mflops

Table 3.4 compares METRON processing overhead against a vanilla system while running using either the system call injection method or the function call injection method to capture tainted values. The Linpack benchmark evaluates the system floating points operation performance by calculating the number of operations the system can execute in a given time. Our results show that our initial approach using system call interception creates a 500 times slowdown on this benchmark, which confirms previous studies observations [4]. However, when run using the function call injection, we observed only a two times slowdown for the instrumented version. This slowdown is actually much lower in common applications: the Linpack micro-benchmark extensively uses numerical operations, while common Android applications, such as pedometers, only have

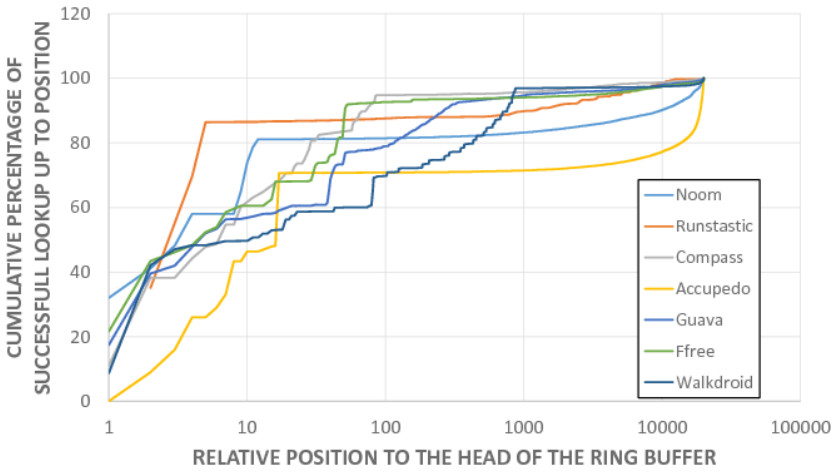


Figure 3.5: Ring Buffer Usage for Several Applications

a very small fraction of numerical operations. In statistics presented in Table 3.1, a general app used 0.42% floating point operations compared to the Linpack's 29.35% in Table 3.4, or an increase of 70% in operations to track. Therefore, the global overhead of generic app revolves around 3%, depending on the ratio of numerical operations they have.

Ring Buffer Lookup Optimizations. Figure 3.5 shows the cumulative percentage of successful lookups on the ringbuffer at each position for a previously recorded tainted value. Between 42-83% of the lookups for tainted values succeed by looking up the ten most recently inserted values in the ring buffer. This justifies the design trade-off described earlier for which a linear search on the few last values inserted is performed first (in this example, on the five most recent values) followed by a tree based search if any of the first values was a match. For this evaluation we used a 20,000-value ring buffer. An average binary tree search in the ring buffer will thus require a maximum of 15 comparisons. The size of the ring buffer we used for our tests was determined empirically by executing METRON and TaintDroid on a same system. We increased the size of the ring buffer until no failed values lookups were detected.

Memory Requirements. METRON globally requires a fixed 100KB of memory per app in order to store the ring buffer. By comparison, solutions like TaintDroid require around 9.5% extra memory by application, depending on the number of variables

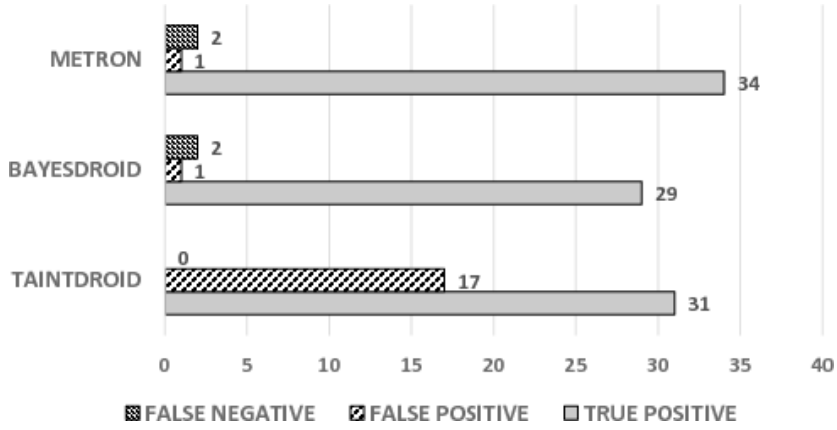


Figure 3.6: DroidBench results summary from METRON, BayesDroid and TaintDroid

allocated.

6.2 Flow Tracking Accuracy

This section evaluates METRON in terms of the information flow detection accuracy. We present two kinds of evaluations. We first evaluate the accuracy of METRON using the DroidBench benchmark by comparing METRON against TaintDroid and BayesDroid. DroidBench provides us with fine grained comparison about the different kinds of information flows that can be detected by each solution. Second, we evaluate the practicality our system running and investigating real applications installed from the Android market.

DroidBench. We ran the set of DroidBench tests presented in the BayesDroid paper. Figure 3.6 shows DroidBench test results for TainDroid and METRON compared to BayesDroid. METRON scored 34 true positives compared to 31 and 29 by TaintDroid and BayesDroid respectively. Also, METRON significantly improved over TaintDroid terms of false positives. Both METRON and BayesDroid reported only one false positive while TaintDroid reported 17 cases. Most of TaintDroid false positives were due to variable sensitivity tests where untainted data was written to variables previously tainted. Both BayesDroid and METRON reported two false negatives. The two false negatives are due to two string obfuscation tests: Loop1 [3] and ImplicitFlow1 [2]. The

Table 3.5: Taint tracking comparison of METRON versus TaintDroid (TD)

App Name	# tot inst.	# num inst.	# TD taint inst.	# Metron taint inst.	Comment
Accupedo	45550000	3935741 (8.64%)	217587	217585 (217585 fp)	TD array false positives
Ffree	22350000	1524497 (6.82%)	22068	20709 (11368 fp)	TD array false positives
Noom	46650000	4536839 (9.73%)	210138	209072 (93714 fp)	TD array false positives
Runtastic	130750000	11304888 (8.65%)	10985	10985 (10985 fp)	Identical flow detection

ImplicitFlow1 test is particularly interesting as it generates two data leaks from a sensitive value using implicit flows by using correspondence tables. The first test obfuscates the sensitive value while the second uses the correspondence table to recreate the original string. Since METRON relies on the representation of the value at the sink in order to detect a flow, this second implicit flow was actually detected. In general, METRON does not support implicit flows. METRON and BayesDroid unique false positive was due to the PrivateDataLeak3 test [1]. This test first write a sensitive value to a file (which is not considered as a sink by the benchmark, but is by METRON and BayesDroid) and then sends the data out via SMS. The legitimate leak happens when the data is sent via SMS. Overall, METRON provides more robust results than TaintDroid while being able to track the flow of numerical values which is not supported by BayesDroid. The detailed evaluation results of DroidBench experiment are shown in Table 3.8.

Apps from Play Store. In addition to the DroidBench detection accuracy, we compared METRON and TaintDroid taint detection while running popular real-world applications downloaded from the Google Play Store. Namely, we chose popular applications that manipulate accelerometer data such as pedometer and fitness tracking applications.

We have ported our value-based information flow tracking implementation of to Android 4.3_r1 in order to run it side-by-side with Taindroid [35] and evaluated both of them in terms of detection accuracy. We investigated how many tainted numerical operations were missed on each side. Table 3.5 presents our results. Several observations can be extracted from this table. First, by comparing the total number of instructions executed versus the number of numerical operations executed we can see that generally this last group represented less than 10% of the total number of instructions. As an accuracy evaluation, we recorded the number of instructions manipulating tainted

Table 3.6: Flow observed in two popular fitness tracking applications

Application name	Flow observed	Notes
Guava	Z accelerometer value saved to file	Raw sensor values leaked
Accupedo	Value `lapdistance` saved to a file	Leaked value computed from multiple sensor measurements

values detected by METRON and TaintDroid.

Overall, we observe that the number of instructions detected by TaintDroid is generally higher than the number of instructions detected by METRON. However by looking closer at individual results, we noticed that the difference between these detection numbers are due to two reasons. First, TaintDroid generated several false positive flows. For the first three applications reported in Table 3.5, TaintDroid raised a false positive alert and METRON did not. These false positive alerts were due to a wrongly tainted array in TaintDroid. TaintDroid naively marks a whole array as tainted even when only one value within the array is tainted. We observed that it is not uncommon for pedometer applications to use arrays to store temporary computation results. Therefore, a false positive was raised whenever any part of the array was sent to a sink.

We have used METRON to examine two popular real-world applications manipulating accelerometer values for information flow tracking. Table 3.6 provides the case of two information leakage cases we captured. METRON also reports the history of operations that were used to compute the leaked value. Hence, it determines whether raw sensor values were leaked or if it is a computed function of it. Listing 3.2 shows a sample output of the detection mechanism at the sink for the Accupedo application. In this sample we can see clearly the contribution of the accelerometer values for the X-axis (denoted with the $S0$ prefix) at several time intervals. Likewise, our investigation of the Guava app revealed flow of values directly recorded from the sensor on the Z-axis.

7 Limitations and Discussion

Non-numerical Data Flows. METRON is designed to track the flow of numerical values alone. While numerical values represent most of the sensitive information (e.g. accelerometer and GPS data), sensitive values can exist in different forms such as strings. Complementary approaches for value-tracking such as *BayesDroid*, which track

Table 3.7: App name to application package name and version correspondance table

App name	App package name	Version
Accupedo	com.corusen.accupedo.te	5.9.8
Ffree	com.ffree.Pedometer	2.6.0
Noom	com.noom.walk	1.4.0
Runtastic	com.runtastic.android.pedometer.lite	1.6.2
Guava	com.guava.pedometer.stepcounter	2.3.0
SensorBox	imoblife.androidsensorbox	5.0
Linpack	com.sqi.linpackbenchmark	1.4

the flow of strings can only be used jointly with METRON to provide exhaustive coverage for value tracking.

Implicit Information Flows, Obfuscation and Encryption. All the three solutions considered: TaintDroid, BayesDroid, and METRON fail to identify information leakage via implicit flows. This chapter also assumes that the values computed are not encoded, obfuscated or encrypted before reaching a sink. Application developers sometimes use encoding protocols such as base64 for convenience. While it would be possible for our solution to detect known encoding and obfuscation schemes, the implementation described in this chapter does not consider this case.

Native Code Support. Similar to TaintDroid and BayesDroid, METRON does not track the flow of information within the native code libraries executed via JNI. However, rewriting the native libraries by injecting additional code to track the propagation of sensitive values through numerical operations within the native library could be considered as an extension to METRON.

Risk Analysis of Information Leakages. The main purpose of METRON to provide insights about the values being leaked by an application. We do not claim to replace the user judgment about whether or not a flow is legitimate as different users may have different sensibility regarding the data they are willing to share. However, METRON is capable of providing the user with a log of the history of operations that were used to compute the leaked values. This operations history can provide a good basis for further research about how to quantify the risk associated with leaked computed values.

8 Related Work

Dynamic Information Flow Tracking. The state of the art in information flow tracking on Android is Taintdroid [35]. In terms of granularity, TaintDroid provides up to a variable-based information flow tracking solution. Its design relies on attaching shadows memory taint values to every variable or object stored in the system memory. TaintDroid, as observed by prior research, suffers from reporting false positive leaks due to variable sensitivity. BayesDroid [87] was proposed as an alternative approach to TaintDroid to reduce the false positives rate. BayesDroid uses Bayesian reasoning and the Hamming distance between strings read at the source and to string detected at the sink in order to detect flows. However, BayesDroid is only limited to tracking string values and does not track dynamically computed values which are covered by METRON. In addition, both TaintDroid and BayesDroid were implemented on top of the Dalvik VM which is now replaced by the ART runtime environment. Therefore, they require OS modifications and are not compatible with newer versions of Android.

Protections against unwanted inferences from sensor data have been presented in previous research such as ipShield [21] which implements a firewall for sensor data. However, such protection mechanism also requires using custom versions of Android.

Both [12, 83] presented preliminary research for how to revive dynamic information tracking under the new Android Runtime (ART) environment. Similar to us, the three solutions modify the `dex2oat` compiler, but both require system modification and cannot work without elevated privileges. Also, we present a way to achieve taint tracking with minimal compiler modifications and therefore make it easier to port our solution to future versions of Android.

Programming languages such as Jeeves [95] enable the developer to define fine-grained information flow rules in the application code. While this approach provides a great control over information flows inside a program, it does not fit most of off-the-shelf applications that are shipped already compiled.

Application Sandboxing Techniques. Boxify [11] and NJSAP [15] demonstrated how to encapsulate the execution of a third-party application with a virtualized environment. While [15] relied , like us, on `ptrace` for system call interception, [15] relied on `libc` function hooking. Our work shows a comparable result in terms of virtualization overhead while extending the sandboxing environment to implement a fully-working value-tracking environment by coordinating a modified version of `dex2oat` compiler. Excuting both techniques rely on system call introspection in order to allow the execution of a third-party application in the context of another application. These solutions provide the advantage of not requiring any system or application changes. Another approach to instrument an application without modifying the system or the application consists of hooking virtual methods. ArtDroid [24] provides such a solution. While this approach can be of some use for the specific task, we opted for the combination of a modified compiler with function call injection to achieve an opcode granularity.

Stricter app sandboxing to protect against private data disclosures have been proposed in FlaskDroid [19] and Saint [70]. They extend Android's permission policies and isolate data. However they do not provide a sufficient level of granularity to track sensor inferences.

9 Conclusion

We presented Metron, a dynamic information flow tracking solution. The novelties of Metron are running at application-level without having to modify the underlying operating system nor require elevated privileges to examine other apps, and utilizing a new approach for tracking information flows based on the values-themselves instead of attaching tainted variables to them. Compared to previous work on dynamic information flow tracking, Metron works on top of the latest commodity Android versions. It also achieves better results than TaintDroid, with fewer false positives, and can handle numerical values which are not covered by BayesDroid.

Table 3.8: Comparison of the METRON performances over other solutions: Taintdroid, and BayesDroid

Benchmark	Algorithm	TP	FP	FN
ActivityCommunication1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle2	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle4	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Library2	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Obfuscation1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
PrivateDataLeak3	METRON	1	1	0
	BayesDroid	1	1	0
	TaintDroid	1	1	0
AnonymousClass1	METRON	2	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
ArrayAccess1	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
ArrayAccess2	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
HashMapAccess1	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
Button1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Button3	METRON	2	0	0
	BayesDroid	2	0	0
	TaintDroid	2	0	0
Ordering1	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	2	0
RegisterGlobal1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
DirectLeak1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
FieldSensitivity2	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
FieldSensitivity3	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
FieldSensitivity4	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0

Benchmark	Algorithm	TP	FP	FN
ImplicitFlow1	METRON	1	0	1
	BayesDroid	0	0	2
	TaintDroid	2	0	0
InheritedObject1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ListAccess1	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
LocationLeak1	METRON	2	0	0
	BayesDroid	0	0	0
	TaintDroid	0	2	0
LocationLeak2	METRON	2	0	0
	BayesDroid	0	0	0
	TaintDroid	0	2	0
Loop1	METRON	0	0	1
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Loop2	METRON	0	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle3	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
MethodOverride1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
ObjectSensitivity1	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	1	0
ObjectSensitivity2	METRON	0	0	0
	BayesDroid	0	0	0
	TaintDroid	0	2	0
Reflection1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection2	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection3	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Reflection4	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
SourceCodeSpecific1	METRON	5	0	0
	BayesDroid	5	0	0
	TaintDroid	5	0	0
StaticInitialization1	METRON	1	0	0
	BayesDroid	1	0	0
	TaintDroid	1	0	0
Total	**Metron**	34	1	2
	BayesDroid	29	1	2
	TaintDroid	31	17	0

Chapter 4

Hardware-Enabled Data Protection

1 Introduction

Healthcare management and delivery costs in developed countries are skyrocketing. In response to this trend, federal agencies have supported diverse lines of applied research in the use of technology for health monitoring and intervention [60, 79]. The intention is to take advantage of the state-of-the-art technologies to compile information about medical health, securely, and in real-time, and thereby, transition from reactive and hospital-centered to preventive, patient-centered and cost-effective health care and management with greater focus on well-being.

More specifically, the deployment of an ideal medical diagnostics solution necessitates meeting three core requirements:

i) portability and low-cost. Ease-of-use and user convenience requires a portable solution so that the users, e.g., elderly patients with regular diagnostic/testing prescriptions, can get themselves tested without having to make hospital visits. Additionally, replacing legacy inexpensive (though sometimes tedious) clinical testing calls for a low-cost solution that can be purchased and used by ordinary civilians;

ii) accuracy and performance. Due to their importance and potential life-changing impact, the correctness of the outcome of medical tests, e.g., HIV tests, is crucial. Furthermore, because of the same reasons, patients are often willing to pay higher cost for more accelerated testing procedures. Consequently, the proposed solution must satisfy both needs.

iii) usable security and privacy guarantees. Based on Gallup Poll for the Institute for Health Freedom [42], 70% of the respondents were concerned about the confidentiality

of their medical records [32]. Potentially sensitive information disclosure may result in undesired consequences such as insurance premium raises and negative social affects.

Mobile-based Point-of-Care (POC) diagnostics by taking advantage of miniaturized devices and mobile technology can dramatically increase the role patients take in their own health care, and consequently reduce health care costs. POC diagnostics refer to in-vitro diagnostic tests that do not require the involvement of laboratory staff and facilities to make results available both to the medical professional and the patient [48, 55, 36]. The possibility of integrating POC systems with mobile platforms has been recently demonstrated through the diagnosis of a series of conditions including vitamin-D deficiency and Kaposi's Sarcoma disease [63, 59, 61]. At the same time, the recently increasing popularity of using information technologies for health care has attracted cyber criminals to this area as well, giving birth to various types of malware and adversarial intrusions against medical critical infrastructures. The number of data breaches across health care sectors have increased by 30% during the last year [53]. As a result, while the availability of mobile-based POC diagnostics systems to the public creates great opportunities in the health care domain, it will be associated with serious privacy and security concerns, due to vulnerabilities on the cyber end.

To address the requirements associated with portability, accuracy and security for mobile-based POC systems, in this chapter, we present MEDSEN, an integrated trustworthy POC diagnostic solution, as a portable device plugged into smartphones, that provides end-users with real-time local medical diagnostics, while maintaining privacy guarantees.

As a running example, in the proposed work, we will focus on a POC system that utilizes impedance cytometry to provide blood cell counting for medical diagnostic and disease staging information. Cytometry and particle quantification have been extensively used for the diagnosis of a wide range of pathological conditions such as cancer and infectious (both viral and bacterial) diseases [47, 65, 23, 62, 89, 88, 58]. Therefore, MEDSEN's design can be broadly applicable to a large number of medical diagnostics devices, as they begin to become integrated with smartphones.

The development of the proposed POC solution consisted of two major tasks: First,

we designed and implemented a microfluidic bio-sensor that utilizes impedance cytometry for biomarker detection. Through innovative hardware-based analog signal encoding scheme, realized via multi-electrode excitation, MEDSEN becomes enable of implementing cryptographic one-time padding encryption to realize trustworthy analog signal encryption.

Second, we designed and deployed a new password and authentication scheme, called cyto-coded scheme, for POC devices. Each password consists of a specific secret ratio of micron-sized synthetic beads, that will be mixed with individual's blood sample. Feeding the mixture to the POC device will constitute the autonomous authentication mechanism without the user's explicit password entry that prevents cyber intruders from accessing patient's sensitive medical results.

The contributions of this chapter are the following:

- We introduce a new portable medical diagnostics solution that leverages smartphone and cloud computational capabilities for heavyweight data processing of bio-sensor measurements.

- We present a domain-specific usable security and authentication technique for POC medical devices via cyto-coded passwords. The proposed scheme removes the need for traditional explicit password entries by the users.

- We propose a user privacy-preserving algorithm for medical healthcare platforms through in-sensor built-in analog signal encryption and decryption schemes. Similar to digital homomorphic cryptographic solutions, the proposed framework allows for cloud-based signal processing on encrypted measurements.

- We have implemented a real-world integrated working prototype of the proposed algorithms and designs. Our empirical evaluation results prove the deployability of MEDSEN in real practical settings.

MEDSEN's sensor encrypts the analog signals using an embedded micro-controller. MEDSEN's software-based diagnostics data processing runs on either Android $v4.1$ (for smaller datasets) or a cloud server that runs Matlab framework.

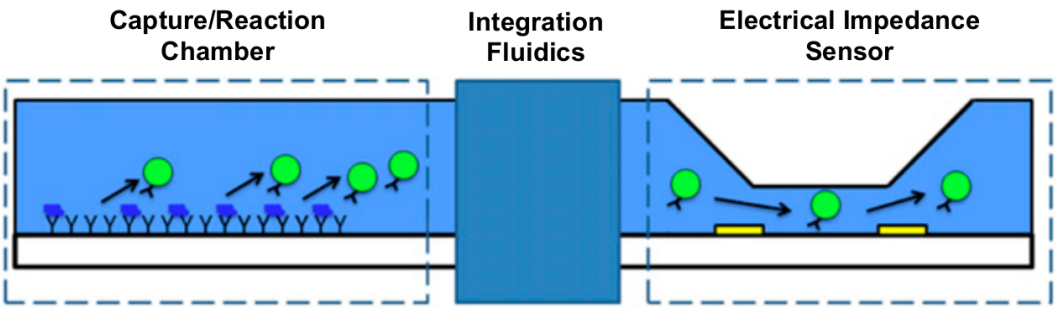


Figure 4.1: Capture Chamber for Cytometry-Based Disease Diagnostics

This chapter is organized as follows. Section 2 presents MEDSEN's components and how they are interconnected logically. Sections 3-5 explain in details the design of MEDSEN's individual components and how they provide their corresponding functionality. Section 6 describes fabrication procedures and details of system integration to implement MEDSEN for disease diagnostics. Section 7 presents our empirical and integrated evaluation of MEDSEN's components in real-world settings including using real blood cells. Section 8 reviews the most related existing work and discusses how MEDSEN addresses their shortcomings in terms of portability, security, and real-time diagnostics. Section 9 concludes the chapter.

2 Overview

MEDSEN provides a new portable privacy-preserving microfluidic biomarker detection sensor for cost-effective human disease diagnosis and management using smartphone computational resources. MEDSEN leverages cytometry [67] to measure the biomarkers in blood. Figure 4.1 shows a functional and low cost implementation for a cytometry framework that uses a probe-molecule (antibodies) coated microfluidic channel to pre-concentrate target biomolecules (cells, viruses, proteins, nucleic acids, etc..) of interest on the channel surface. These specifically bound cells are then released from the surface and then flow though an electrical impedance sensor, where they are singly counted when passing through a set of electrodes. The total electrical cell count is proportional to the total concentration of target biomolecules that were present in the test sample. The peak detection typically requires a software-based implementation of

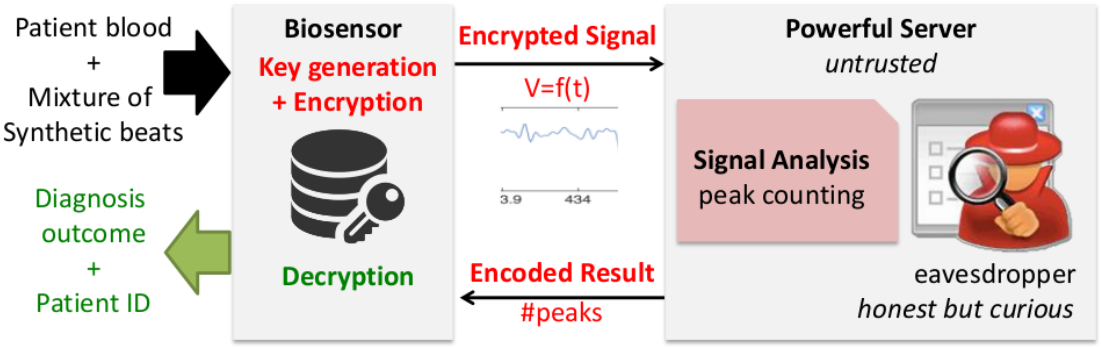


Figure 4.2: Blood sample (<0.01 mL) is drawn from patient and injected into the bio-sensor, which detects the cells and encrypts the data. A powerful server, used to quantify data, will obtain a convoluted result. The microprocessor integrated with the bio-sensor will obtain the convoluted results and use a unique key sequence to decrypt the encoded result.

signal processing for denoising and removal of baseline drift and peak detection.

Figure 4.2 shows MEDSEN's high-level architecture. In addition to biomolecular analysis and disease diagnosis, MEDSEN protects user privacy, while enabling secure authentication to a remote healthcare database server. The patient collects a blood sample using a specifically crafted mini-pipette. MEDSEN feeds the blood sample to the biomarker sensor device that is attached to the smartphone. During data acquisition, the sensor dynamically changes its configuration (i.e., number of active electrodes) to encrypt the outgoing analog signal to the smartphone. Each cell that passes through a pair of active electrodes causes a peak in the measured signal due to the inter-electrode dielectric characteristic changes. The smartphone sends the encrypted measurements to a remote powerful server for disease diagnosis analysis (Figure 4.2).

The server analyzes the signals and counts the number of peaks, which does not necessarily correspond to the true number of cells/biomolecules/beads that were present, because more than one electrode pair may have been activated during data acquisition. The number of activated electrodes is confidential to MEDSEN's sensor device. The server sends the counted number of peaks back to the MEDSEN sensor for decoding. MEDSEN simply decodes the number and determines the user's disease condition through a simple threshold comparison, and notifies the user accordingly. MEDSEN's in-sensor analog signal encryption, as opposed to traditional cryptographic digital data

point encryption, eliminates the need for complicated analog-digital conversion circuitry and powerful cryptographic encrypting processors within the sensor.

For authentication, the user's blood sample is mixed with a user-specific number of artificial beads before passing through the MEDSEN's sensor (Figure 4.2). The peaks caused by the artificial beads differ in amplitude from the peaks caused by the true biomarker cells and can be distinguished on the server side. Based on the number of counted artificial beads, the server authenticates the user (similar to password checking) without the need to explicitly screen password entry by the user. Consequently, the diagnostic information can be returned to a patient or stored in cloud for a later access by the patient's practitioner.

Threat model. We have microfabricated a multi-electrode cytometer with the ability to obtain, modulate and obfuscate peak number during data acquisition. A cryptographic algorithm is imposed on the bio-sensor via electrode key multiplexing, making the true number of peaks unattainable to potential eavesdroppers who do not have the appropriate security key. The proposed method allows for elevated security in diagnostic devices against confidentiality attacks when transmitting sensitive medical data over the network or their processing in the cloud. Contrary to previous work that uses encryption or authentication as an independent component in a system design, our encryption is embedded in the cytometry operations. This close coupling between the signal acquisition and the encryption process allows our setup to have a very small trusted computing base (TCB). MEDSEN's trusted computing base is its sensor. Aside from the sensor, which physically manipulates the patient blood sample, and the combination of a small controller and a multiplexer responsible for managing the diagnostic experiment settings (electrodes voltage and current), no other component has access to the true cytometry information. MEDSEN neither trusts the smartphone nor the remote server, because they both see only the encrypted measurements and the analysis outcomes. Those parties are assumed to follow a curious but honest adversarial model.

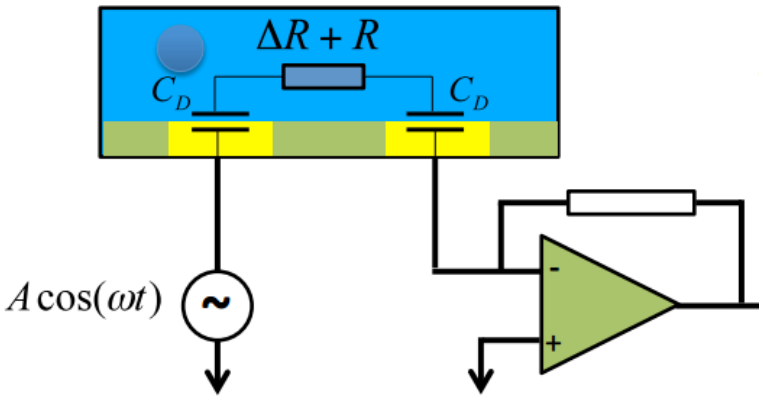


Figure 4.3: Model of operation of planar electrode pair. The electrode-electrolyte interface is modeled by the double layer capacitance. The electrical impedance in the channel fluctuates as the cell/bead passing between the measurement electrodes.

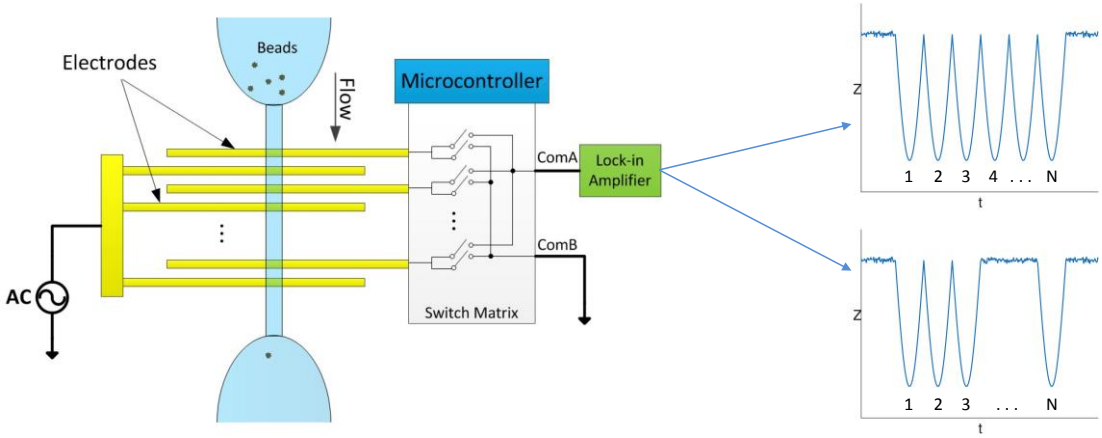


Figure 4.4: Operation model of the integrated system. Input electrodes connected to AC voltage source. Ouptut electrodes connected to analog switch controlled by microprocessor. Controller randomly activates different subsets of electrodes, resulting in multiple peaks for each cell detected.

3 Medsen System Design

3.1 Bio-sensor

MEDSEN bio-sensor is integrated in its microfluidic system and acquires data by monitoring the electrical impedance across the channel. Figure 4.3 shows MEDSEN's electrical impedance measurement setup in the microfluidic channel that consists of co-planar electrodes. The electrical impedance of a bead (micro-particle) passing through the microfluidic channel is detected by changes in measured impedance between the electrode pair (capacitor). The input electrode is excited with a continuous AC signal at a fixed frequency. The output electrodes are connected to a lock-in-amplifer that converts the current to voltage, and locks into the AC excitation frequency. As the bead passes between the electrode pair, the electrical impedance increases because of a partial occlusion of ions passing between the two electrodes. Thus, changes in voltage at the output of the lock-in-amplifer correspond to the beads being detected.

Figure 4.3 shows how the sensing electrode pair in the microfluidic channel can be modeled as a series of capacitors and resistors [33]. The resistor depicts the resistance of fluid and contents passing through MEDSEN's microfluidic channel. The parasitic capacitance results from a double layer of ions forming at the electrode, i.e., the electrolyte interface. When voltage applied to the electrode, a layer of ions polarity accumulates at the surface of electrode. The electric field from the electrodes is screened by the free ions in the double layer, similar to a parallel plate capacitor. The system of capacitors and resistors in series will have a distinctive capacitance dominant region and resistance dominant region in response to a range of applied frequencies. At low frequencies ($<$10 kHz), the system response is dominated by the electrical characteristic of capacitance, and thus the measured impedance is relatively high (MΩ range). At higher frequencies ($>$100 kHz), the capacitance is short circuited. This prompts the resistance to dominate the impedance. We desire to operate in the regime where resistance is dominant since we are measuring changes in ionic resistance resulting from the presence of beads in between the electrodes. Each bead or cell passing by results in a single peak in the output voltage.

3.2 Multi-Electrode Signal Encryption

MEDSEN expands the simple impedance cytometer and uses multiple electrodes with multiple inputs shorted together and multiple independent outputs. This results in multiple peaks as each cell or particle passes by. The individual outputs are used as the key component in our biomedical microelectromechanical system (BioMEMS)-based signal encryption. The output of the electrodes can be selected or discarded through the multiplexer by the (pseudo-)random selection of a micro-controller. The signals of the independent output electrodes can be randomly switched on or off through a multiplexer chip. The details of signal encryption using random keying of output electrodes are described in Section 4. The repeated readings of cell impedance of a single cell when passing through the electrode pairs in the bio-sensor can be manipulated to a random sequence of peaks response by arbitrarily selecting the outputs from the electrodes pairs. Figure 4.4 describes the operation model of the integrated system. This results in a randomly varying number of peaks for each bead passing by. A potential eavesdropper, without access to the signal encryption key, will not be able to discern the true number of beads that have passed by. The number of beads is a crucial parameter for disease diagnostics analyses. For instance, the white blood CD-4 cell count is the strongest predictor of human immunodeficiency virus (HIV) progression in lab tests nowadays.

We designed the microfluidic channel with integrated multi-electrode pair configuration to mask the number of peak count for passing cells in microfluidic channel to protect user privacy and security in diagnostic devices against confidentiality attacks. Figure 4.5 shows the computed aided design (CAD) of the microfluidic bio-sensor. Figures 4.5a and 4.5b show the designs of multiple bio-sensors embedded along a single microfluidic channel. Figure 4.5a shows the designs of two sets of sensing electrodes. On the left side, the bio-sensor has two independent outputs; on the right side, the bio-sensor has three independent outputs. Similarly, Figure 4.5b describes the design of bio-sensors with five and nine independent outputs on the left and right sides, respectively. The input excitation to all electrode pairs in each sensing region are tied together to a common excitation source. The analog front-end circuitry connected to

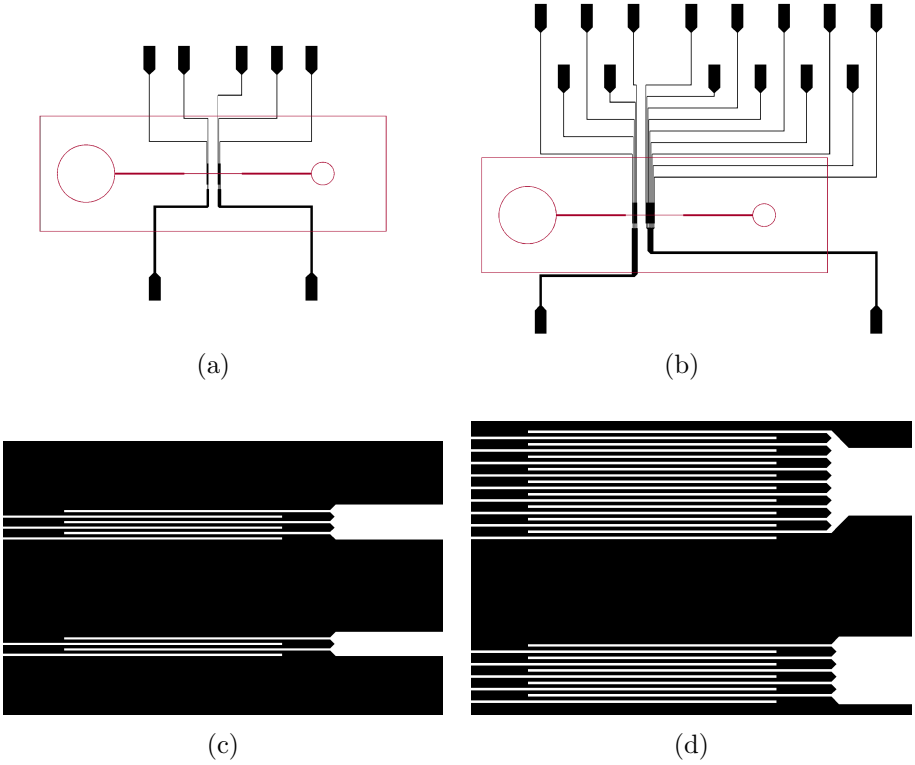


Figure 4.5: Design of the electrodes. (a) Bio-sensor design with 2 outputs (left) and 3 outputs (right). (b) Bio-sensor design with 5 outputs (left) and 9 outputs (right). (c) Details of the sensing regions of 2 and 3 output electrodes. (d) Details of the sensing regions of 5 and 9 output electrodes. The red outlines on the bio-sensors depict the microfluidic channels.

the output electrodes captures the current change between the electrode pairs in microfluidic channel independently as a cell passes through. Figures 4.5c and 4.5d show the details of the active regions each sensor in Figures 4.5a and 4.5b, respectively. The *lead* electrode in the array of output electrodes, is defined as the lower left electrode in each sensing region. The lead electrode is only complemented by one input electrode on its right side. Thus, it will respond with a single voltage drop per passing cell; whereas the remaining output electrodes are surrounding on both sides by common excitation electrodes. Each of the remaining electrodes in the sensing regions will respond with a signature of double peak per passing cell.

3.3 Microfluidic Channel

For evaluation and testing of the sensing platform, we analyze the impedance of different synthetic bead types (7.8 µm and 3.58 µm) and the blood cells passing though electrode pairs embedded in the microfluidic channel. These specific bead sizes are chosen as they approximate the dimension of various cells found in human blood. The microfluidic channel is designed to accommodate the transport of blood cells and beads passing through electrode pairs. In the interest of counting and modulating the number of cell counts passing in the microfluidic channel, the channel dimension were designed to pass a single bead or cell through electrode pairs one at a time. Figure 4.6 describes the design of microfluidic channel. The measurement pore, which is the narrow channel at center, helps to single out and deliver synthetic beads and blood cells in succession. The wide regions at both ends of the measurement pore allow the beads or blood cells to disperse before entering the measurement pore of microfluidic channel. The two circles depict the inlet and outlet of the channel after the PDMS is removed using biopsy punchers.

4 Sensor-Based Analog Signal Encryption

We describe how MEDSEN encrypts cell signal measurements. It uses a symmetric analog encryption scheme that relies on the choice and secrecy of a key to protect

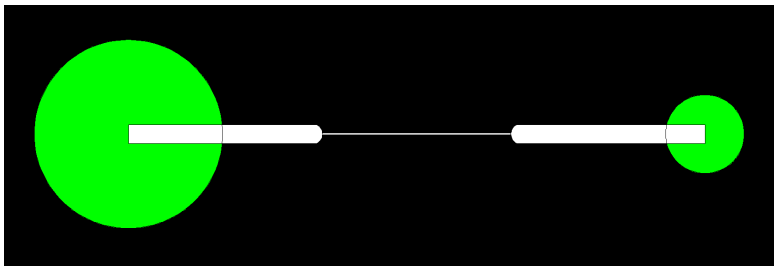


Figure 4.6: Microfluidic channel design. The measurement pore (thinner channel at center) has width of 30 µm and length of 500 µm. The larger regions at both ends of the measurement pore allow the beads/cells to disperse before entering the measurement pore. The two circles depict the inlet and outlet of the microfluidic channel after the PDMS is remove with biopsy punchers.

the encrypted measurements. The encryption operations are embedded in the sensor itself and infer no overhead for the encryption operation in terms of time overhead. To enable encryption, the sensor components have been specifically crafted using multiple electrodes to modulate peak counts generated by cells, such that no external entity can recover knowledge of the number of cells passing through the channel from a specific number of peaks on a signal acquired. This encryption scheme can also alter the measured signal peaks' amplitude and width such that the resulting encrypted signal can be analyzed by an untrusted third party processing resource, like a cloud service, without revealing any useful cytometry information. Eventually, only the bio-sensor in possession of the patient, can decipher the information carried in the analyzed signals based on the randomly generated key that it had generated initially.

4.1 Cipher Design

The strength of MEDSEN's signal encryption methodology relies on its bio-sensor's reconfigurability to generate various signal measurements, possibly with different amplitudes and shapes for a single cell passing through the channel. The sensor configuration is determined dynamically by the randomly generated key on MEDSEN's bio-sensor micro-controller. MEDSEN's sensor design hides the information carried by a signal from the external untrusted entities by generating multiple signal peaks of different shapes.

A cell passing through electrodes consistently generates a voltage drop between the electrodes. Figure 4.7 shows such a variation in our empirical experiments. The peak

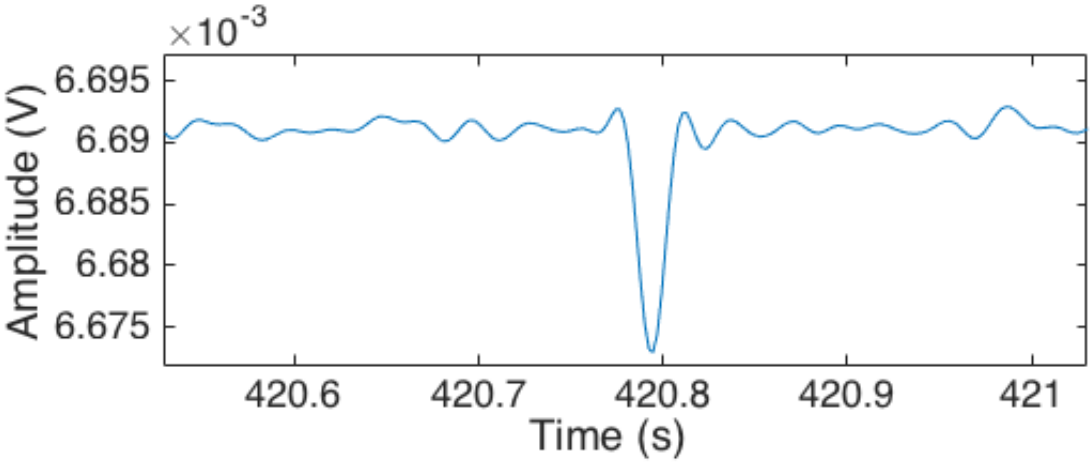


Figure 4.7: Voltage Drop When a Cell is Passing Through the Electrodes

used to infer a diagnosis (Section 2). Our cipher leverages a specific sensor design and a custom protocol to multiply and transform a signal acquired from a single cell into a random sequence of signals unrelated to the cell properties. Only the random sequence issued by the micro-controller, which defines the sensor configuration can decrypt the values behind the sensor measurements. To randomly clone a single peak signal into multiple peaks signal, the sensor activates and uses multiple electrodes that are selectively powered on or off in such way that the bio-sensor generates a random succession of electrode order. The response of such electrodes configuration will precipitate signal with peak count number output larger than the actual number of cells passing through the micro channel. The resulting multi-peak signal conceals the actual number of cells passing through the channel. For example, Figure 4.8 shows the resulting signal of a single blood cell detected by MEDSEN, resulting in a five-peak signature for a single cell.

This resulting signal hides the number of cells, but still carries information about the cells. Specifically, the amplitude or the width of a voltage drop can reveal information about the composition or shape of the cell. To protect both information, MEDSEN cipher design leverages two more parameters to protect this information. First, randomly chosen voltage gains can be applied by electrode such that none of the peaks carries the amplitude drop of the original signal. This gain information is incorporated as part of the encryption key. Similarly, a modification of the flow speed on the channel would

result in peaks of arbitrary widths for cells of identical type. By leveraging these three parameters, the number of active electrodes, the electrodes gain and the fluid flow speed on the channel, the controller can generate any number of peaks with any shape. These transformations allow the sensor to conceal the sensitive cell information and to later recover them thanks to the parameters embedded in the key.

The specific sequence of electrodes turned on or off, the set of output gains applied to electrodes and the fluid flow speed in the micro channel constitute the encryption key of the biomarker measurement signal. To preserve the initial signal's confidentiality, every peak p associated to a cell would have a different set of chosen parameters, or key K_p, such that:

$$K_p = (E_p, G_p, S_p) \tag{4.1}$$

with E_p is the binary vector representing the sequence of on/off electrodes, G_p is the sequence of electrodes gains, and S_p represents flow speed on the channel. Such a design choice would lead to a key size of length $L = cK_p$ for c number of cells passing through the channel. Such a setup is comparable to the perfectly secret one-time pad encryption scheme [73]; every signal peak is encrypted with its own randomly generated key. Accordingly, the key length varies linearly as function of the number of cells. Such an encryption algorithm would ensure a perfectly secret encryption since it can produce any resulting shape for a given original signal. In practice, applying a different set of parameters per cell measurement is challenging as it increases the key size, and would require MEDSEN to be aware of every cell entering and leaving the channel. Moreover, we observed that multiple cells can pass through the channel with a distance interval inferior to the distance between the first and last electrode. Thus, two or more cells may appear among the electrodes simultaneously; this complicates the signal encryption and decryption procedures. Consequently, MEDSEN implements an alternative scheme that periodically changes the encryption parameters every time unit: $K(t) = (E(t), G(t), S(t))$.

This cipher has the key characteristic that the encrypted signal can still be processed to detect voltage peaks. A peak detection algorithm (Section 6.1) can be performed

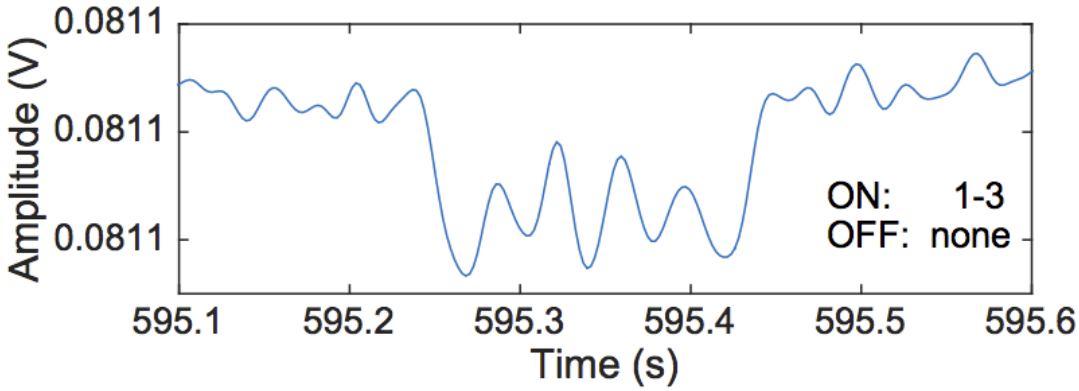


Figure 4.8: Encrypted cytometry Signal for a Single Blood Cell. Output electrodes 1-3 turned on by switch matrix results in five peaks due to one cell passing by the sensor.

on the encrypted signal and returns encoded peak count, with associated time-stamps, amplitudes and widths. Given a ciphertext, it is impossible for a domain knowledgeable attacker to infer the patient diagnostic. Only the controller, which knows the input values applied to each control parameter, is able to recover the real signal amplitude and cell count associated to the ciphertext signal peaks. It is noteworthy that the presented encryption scheme do not infer any noticeable encryption computation overhead or delay since it is based only on hardware configuration, built in the sensor. The decryption requires light computation (multiplications and divisions) and can be performed on MEDSEN's resource-constrained controller.

MEDSEN hides the number of cells with the multiple electrodes sensor deployment that generates multiple peaks per cell. So, the attacker cannot recover the number of cells captured in the chamber. A determined attacker would then try to recover the number of electrodes turned on and off in a particular channel to recover the peak multiplication factor generated by the multiple electrodes. By dividing the number of peaks observed in a data set by the multiplication factor, the attacker would recover the initial number of cell passing through the channel. Considering that each cell has a specific signature in term of voltage drop when passing through a set of electrodes, the attacker would try to detect consecutive peaks of the exact same amplitude and then infer the number of electrodes on. The cipher design protects this information by applying random gains on each electrode output. This changes the signal amplitude

and thus conceals the initial signal characteristic. Similarly, an attacker could try to recognize peaks that correspond to a single cell by observing the width of the curve that would remain unchanged by modifying the amplitude. By modifying the fluid flow speed through the channel, MEDSEN can alter the width of the resulting signal and thus protect this information as well. The slow fluid speed results in peaks with larger widths.

4.2 Cipher Key Space Size Analysis

The encryption key chosen to encrypt the original signal corresponds to different design parameters. All combined, these parameters define possible combinations to obfuscate the signal. These parameters must be chosen carefully so that the resulting signal remains detectable and measurable without introducing an attack surface for bruteforce password attacks because of a limited key space size.

Minimum and maximum measurable signal amplitude The key space and the entropy of the cipher is closely related to the *signal to noise ratio* that the sensor can achieve. Intuitively, if an electrode gain is so large that it "buries" another electrode signal in its noise, the peak recognition as well as the decryption would fail. In practice, this is not always true, since multiple measurements allow some redundancy and permit to decrypt the signal even with high noise. We here provide a conservative analysis (ignoring extra information due to multiple measurements) of the entropy of this cipher.

Condition C1: *The minimum gain applied to an electrode during one period multiplied by the minimum peak size must be superior to the maximum gain applied multiplied by the noise amplitude for the same period.*

Practically, the decryption phase considers signal peaks that exceed a predefined threshold above the average. The electrode encryption gains must be chosen such that no encrypted signal has a cumulative gain on the electrodes too low in comparison to other periods. This results in a constraint on having uniform cumulative gains across various periods, which makes the encrypted signal indistinguishable to the adversaries. Consequently, it becomes harder to detect a change based on the average noise over

time periods.

Condition C2: *The sum of the gains of the electrodes over various signal periods must add up to a single value throughout the experiment.*

The feasible amplitude range is dictated by the circuit's physical constraints, i.e., *minimun_signal* vs *maximum_noise*. The signal to noise ratio impacts the amount of entropy attainable per encrypted peak.

Encryption time period duration in addition to the experiment parameters that obfuscate the signal (multiple electrodes, gains, flow speed), the encryption key can be changed periodically in order to add entropy to the obfuscated signal. This rekeying allows a better protection by obfuscating the original signal following different parameters over time. The duration of the key period must be chosen carefully. In an ideal scenario, the time period would correspond to the time for a cell to transit across all of the electrodes assuming that cells are passing through the channel at a constant interval rate. In practice, the period depends on the concentration of cells and the time length of the measurement corresponding to one cell passing. The encryption period must be defined such that it is not too long, because otherwise, an extrapolation of the result over the cell transit period would lead to the diagnostic false positive. The period should not be too short either, since a period switch might happen during a single measurement and generate a distorted signal caused by the same single cell.

Formulation of the key length in function of the experiment parameters we provide below a calculation of the number of possible key combinations that can be used for encryption by combining the different experiment parameters. The number of key combination possible by simply turning electrodes on or off for a n electrode pair sensor is 2^{n-1}, assuming that zero electrode on is not a valid combination and that shifted combinations are equivalent (e.g.: on a four electrode sensor, 1 denoting a pair of active electrode and 0 denoting a pair of electrode off, 0101 is the same as 1010, 1100 is the same as 0110 and 0011, etc). Moreover, assuming that m different levels of gains can be applied per electrode pair and that the same gain is not used twice on two

electrodes pairs in a same combination, we obtain a number of possible key combination equal to:

$$\sum_{k=0}^{n-1} \frac{(n-1)!}{k!(n-1-k)!} \frac{m!}{(m-k-1)!}$$

where $\frac{(n-1)!}{k!(n-1-k)!}$ corresponds to the number of different combination possible with $k+1$ electrode(s) on for n electrodes sensor and $\frac{m!}{(m-k-1)!}$ corresponds to the number of ways of choosing $k+1$ different gains levels for these pair of electrodes on.

In addition to the electrode pattern and gains applied per electrode a rekeying happens periodically and a different flow speed is used at each one of this phase.

5 Cyto-Coded Authentication

Cloud-based medical services often require user authentication for various reasons such as billing and/or data storage for later remote access by the users' doctors. For server-side patient authentication, existing solutions leverage traditional methods such as explicit on-screen password entry by the user [74]. MEDSEN's trusted computing base does not include the user's smartphone; hence, it cannot rely on phone-based user credential entry. On the other end, adding password entry screen and authentication data processing facilities would increase MEDSEN's complexity, size and cost. Instead, MEDSEN eliminates the need for explicit password entry completely, and makes use of the user's blood entry channel for automated authentication of the user. We introduce a new authentication mechanism that rely on unique identifier based on a new type of alphabet.

This alphabet is build by choosing unique combinations of types and quantities of beads that are mixed with the patient blood sample. These micro-beads generate peaks in the ciphertext signal, similar to blood cells with different peak characteristics, that can be recovered by the controller after the decryption stage and used to associate a stored ciphertext in the cloud service to an unique bead-based identifier that only the patient possess. This cyto-coded identifier protects the patient privacy by default since it is embedded in the diagnostic protocol and carries no biometric information.

It can be associated either to a single diagnostic (different identifiers per pipette), several diagnostics (multiple pipettes carrying the same identifier) or the entire set of diagnostics from a specific user (all pipettes from a user) depending on the diagnostic privacy requirements. Also, it is not linked with any patient related information or knowledge, relieving him or her from the task of ensuring the privacy of his or her data.

Also, this identifier permits to check the integrity of the ciphertext. More precisely, it provides a verification code to ensure that the integrity of the signal processing. The results returned by the cloud-based server is preserved if the decoded synthetic bead types numbers matches the ones submitted initially. If the identifier recovered from the ciphertext differs from the the one used to fetch the data from the remote service, then the ciphertext is not the one corresponding to the identifier. Section 6 and Section 7 describe how to chose the bead types and concentrations in order to generate a dictionary of unique identifiers with limited risk of collisions of passwords by different users.

Additionally, the abovementioned cyto-coded identifier could also be used for server-side user authentication. The bead sample (cyto-coded identifier) is fed to MEDSEN's bio-sensor with the bio-sensor level encryption turned off such that the server-side can reconginize the actual number and types of the submitted beads for authentication purposes. Consequently, the proposed cyto-coded identifier could be leveraged differently based on the specific use-case, where MEDSEN is being deployed for.

6 Implementation

6.1 Medsen Bio-Sensor Fabrication

MEDSEN's microfluidic channel master mold is fabricated on a silicon (Si) substrate using standard process of soft-lithography [93]. The photomask for the photolithography process is design in AutoCAD and fabricated by Advance Reproductions Corp. (North Andover, MA). The silicon substrate is cleaned in acetone and methanol baths using ultrasonic cleaner before fabrication. To fabricate the mold, the Si wafer is coated coated with SU-8 photoresist (MicroChem Inc., Westborough, MA) and exposed to UV

light under the photomask to create the molding patterns. The coated wafer is then developed in MicroChem's SU-8 developer and baked to harden the micro-patterns on the wafer.

Microfluidic channel. MEDSEN's microfluidic channel is constructed with poly-dimethylsiloxance (PDMS) using standard molding techniques [93]. PDMS is an alternative material to silicon micro-machining for fabricating microfluidic channels enabling low-cost rapid prototyping. One of the most important characteristics of PDMS is the optical transparency of the material. The PDMS elastomer is transparent under optical frequencies. Micro-fabricated channels are inspected both visually or under a microscope. Additionally, PDMS can be covalently bonded to Si, glass substrates or to PDMS itself by oxygen plasma treatment. This allows the fabrication of multi-layer structures in microfluidic systems [54]. The major advantages of PDMS in microfluidic system construction is the inexpensive process and rapid fabrication of devices. Microfluidic channels can be cast repeatedly using a master mold. The elasticity of PDMS allows casting of the devices to release the master mold without damaging it. Thus, a single master mold can be used in mass production of microfluidic channels. Furthermore, the construction of simple microfluidic channels can be done in one single casting stage of PDMS as opposed to multiple steps required with silicon and glass micromachining techniques. This channel fabrication technique reduces the time of microfluidic channel construction.

A microfluidic channel with dimension of 30 µm width and 20 µm height is cast using master mold as designed. Sylgard® 184 silicone elastomer base and curing agent (Dow Corning) are mixed uniformly at 10:1 in weight ratio to produce PDMS solution. To create the channel features, the solution is slowly poured on top of the mold to establish conformal contact with the mold. The mixture is cured in 80 °C before peeling off to make microfluidic channel. Due to the high viscosity of elastomer, microfluidic channels take on the designed configurations of the master mold. Biopsy punches are used to create inlet and out outlet ports for the microfluidic channel.

Micro-electrode fabrication. Micro-electrodes are fabricated on the glass substrate using standard photolithography [93]. Similar to previous fabrication of the mold, the glass substrate is coated with photoresist AZ5214 (MicroChem, Westborough, MA) and exposed to UV light under photomask for the electrodes. The coated wafer is then developed in AZ5412 developer to create the micro-patterns for the sensor. A thin layer of chromium and gold (50 Å and 100 nm respectively) are deposited on the micro-patterns using electron beam evaporation to create the sensor. In this design, MEDSEN's electrodes have width of 20 µm, and 25 µm pitch. Figure 4.5 shows the electrodes that are designed with one common rake of electrodes. This common junction takes excitation inputs of multiple carrier frequencies from the lock-in amplifier. The output electrodes are interpolated in between the electrodes of the common junction.

Microfluidic device. PDMS can be covalently bonded to glass or Si substrates by oxidizing the contact surfaces. Cross-linked polymer exposed to oxygen plasma generates a thin layer of silanol terminations (SiOH) on the surface. When brought in contact to the oxidized glass surface, the silanol terminated layers condense with each other. The reaction creates conformal Si-O-Si bonds between the polymer and glass [30]. These covalent bonds create an irreversible, water tight seal between the layers for microfluidic channels. Furthermore, Oxygen plasma treatment on PDMS changes its surface properties from hydrophobic to hydrophilic [85]. Intrinsically, cross-linked PDMS is hydrophobic. Hydrophobicity in microfluidic systems would make it difficult to wet the channels. The introduction of the polar function silanol group by exposing the microfluidic channel to Oxygen plasma renders the surface of the channel hydrophilic. The change from hydrophobicity to hydrophilicity of PDMS is observed by the relative change in the advancing contact angle of deionized water and PDMS surface [50].

6.2 Sensor-Side Data Manipulation

We implemented MEDSEN's peak count-based encryption scheme using the multi-electrode sensor, a controller for random key generation, and a multiplexer for converting the key to a specific sequence of electrodes.

We used a Raspberry Pi as a controller, which is in charge of generating the key that is later used to set a specific sequence of electrodes on or off. For our proof-of-concept demonstrations, we used the controller's Linux operating system `/dev/random` interface as the entropy source for the key sequence parameters. Based on the key, the electrodes are turned on and off via the Raspberry Pi general purpose input-output (GPIO) interfaces that are connected to a multiplexer, which in turn is connected to the electrodes. The key generation and key renewal over time is handled by a python custom library that is embedded in the program. The encryption keys always remain on the controller and never get sent out to the phone or cloud. This keeps the controller as MEDSEN's minimal trusted computing base.

To ensure accurate sensor cell count repeatability and device fidelity, the blood sample evaluated should reach a certain size in terms of the number of cells counted. From repeated experimentation, we empirically determined that samples containing at least 20K cells can provide repeatable cell count with minimal standard deviation from run to run using MEDSEN sensor. For the ideal encryption design (Section 4), i.e. one different key for individual successive cells, this would result in a key of length L bits:

$$L = N_{\texttt{cells}} \times (N_{\texttt{elec}} + \frac{N_{\texttt{elec}}}{2} \times R_{\texttt{gain}} + R_{\texttt{flow}}), \tag{4.2}$$

where $N_{\texttt{cells}}$ represents the number of cells in the blood sample; $N_{\texttt{elec}}$ is the number of activated electrodes; $R_{\texttt{gain}}$ the representation bit-length (or resolution of the gain in bits) for subsequent pairs of activated electrodes $N_{\texttt{elec}}/2$ each forming a capacitor; and $R_{\texttt{flow}}$ the representation (or resolution) of the flow speed in the channel.

Considering a $20K$-cell sample, with a 16 output electrode bio-sensor, with 16 different choices of gains (4-bit representation) and 16 different flow speeds, that would lead us to a $20K * (16 + 8 * 4 + 4) = 1M$-bits key (0.12MB). The 16 different gain and flow speed resolution granularity are empirical choices and can be adjusted based on the security and sensor precision requirements. We made those choices based on the following observations. The amplitude and width of a peak associated with a signal will

typically be as much as four times larger than the smallest peak observable. Specifically, let's take the 3.58 µm synthetic beads as a reference. Human blood cells will typically have peaks of approximately twice the amplitude, and the 7.8 µm synthetic beads approximately have four times the amplitude of the 3.58 µm beads. Choosing a 16-level granularity provides MEDSEN with (more than) sufficient entropy and flexibility to change peak characteristics in order to conceal cell types and masquerade them to external untrusted entities such as the cloud. Needless to mention, higher granularity would help to improve the homogeneity of the signals in the ciphertext and thus provide better protection at the cost of larger key size.

Cyto-coded authentication. MEDSEN's cyto-coded identifiers and authentication alphabet relies on different types of beads mixed together at different concentrations. These two features are used together to provide an alphabet large enough such that we can create large number of distinguishable identifiers for different patients. To avoid an identifier collisions, i.e. two different sets of beads types and concentrations that result in the same measured/classified identifier, we carefully chose different types of beads as well as specific bead concentrations that provide a measurement resolution good enough to avoid any undesired case. This is tightly linked to the sensor precision. Section 7 provides further empirical details on this identifier construction and resolution. From the patient perspective, this approach provides a completely transparent and privacy preserving authentication mechanism. This feature can be especially useful for patients that use this diagnostic framework multiple times such as daily medical tests by the elderly. A set of miniaturized micro-pipettes purchased by the same user would embed the same identifier. Patients do not need to enter any information such as their credentials on the phone or controller. MEDSEN considers the identifier as the patient's credential, and stores the analysis outcomes from the same patient under the same class.

6.3 Cloud-Based Data Analysis

MEDSEN encrypts the acquisition signal by randomizing the number of electrodes (peaks) according to the generated key. For server-side encrypted signal analysis, MEDSEN implements a peak counting method to extract the number of peaks and their characteristics in the encrypted signal. The outcomes are sent back to MEDSEN's bio-sensor micro-controller for decoding and diagnostic decisions. Peak detection is performed by thresholding the acquired signal and computing the number of peaks whose amplitudes are above a predefined minimum value threshold. However, in the long succession of data acquisition, the measured signal changes in the baseline measurement. These changes can be caused by many conditions such as the change in fluid concentration over long acquisition time and the temperature drift of the fluid.

In order to perform peak detection, the signal needs to be detrended before thresholding the signal. Signal detrending is achieved by fitting the polynomial to the signal and detrending the signal according to the polynomial line. Strategically, higher order polynomial fitting is desired to match the baseline drifting of the signal. However, for the large sequence of data, the high order of the polynomial fitting would cause additional unwanted effects such as over-fitting. This would cause the peaks of the signal to deform to a larger degree. For lower order of polynomial fitting, the fitted line might not be conformal to the baseline drifting of the signal. This would be under-fitting and the signal cannot be detrended as desired.

According to our repeated experimentation, we empirically found optimal a second order polynomial fitting line to detrend the baseline drifting of the signal. For the large sequence of the signal, a second order polynomial line clearly under-fits the baseline drift of the signal. However, by partitioning the signal sequence into a smaller train of data sub-sequences, the second order polynomial fitting line would be sufficient to conform the baseline drifting of each section in the train set of the signal. The detrended sub-sequences will be concatenated to create the original signal sequence with signal detrending applied. After fitting the sub-sequence with a second order polynomial, the data section is detrended and normalized by dividing the subsection of data by the

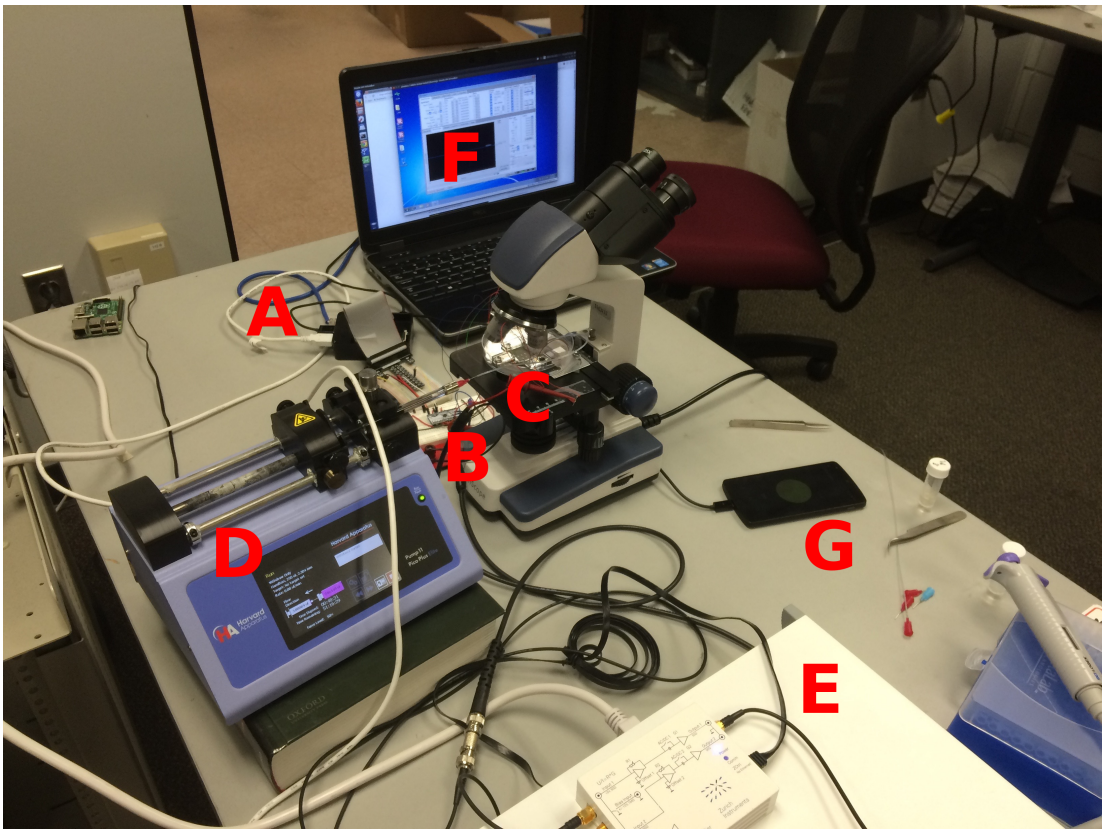


Figure 4.9: Full Experiment Setup: **A-**microcontroller, **B-**multiplexer, **C-**microfluidic device, **D-**external peristaltic pump, **E-**lock-in amplifier. **F-**cloud infrastructure, **G-**mobile platform.

fitted polynomial. The sub-sequences of the signal are detrended with overlap sections to minimize the error of the fitted polynomial at both ends of the sub-sequences. The baseline of the detrended sub-sequences has a mean value of one. Peak detection is achieved by setting a minimum threshold on the data section of one minus the detrended subsequence.

6.4 System Integration

Figure 4.9 describes the implementation of the MEDSEN's proof-of-concept apparatus. Figure 4.10 shows the details of the microfluidic device and its interactions with the local micro-controller, smartphone, and the cloud server. Figure 4.10a shows MEDSEN withdrawing the fluid solution from the microfluidic channel through the outlet of the microfluidic device. The microfluidic channel is video recorded under a microscope

for observing the ground truth (for our experiments) and validation of the recovered signals. Figure 4.10b shows the setup for how the output electrodes of the bio-sensor are connected to the input of the multiplexer. Figure 4.10c and Figure 4.10d show the details of the fabricated micro-electrodes and the embedded micro-electrodes in the microfluidic channel (the two parallel lines run perpendicular to the set of electrodes). The microfluidic channel flow is driven by the external peristaltic pump labeled **D**, i.e., Harvard Apparatus 11 Pico Plus Elite (Figure 4.9). The Raspberry Pi microcontroller (label **A**) is used to generate the random selection sequence of the output electrodes in the microfluidic device (label **C**) through the 16:2 multiplexer MAX14661 (labeled **B**; Maxim Integrated). The selected output sequence of the signal is recovered by the lock-in amplifier (labeled **E**).

To upload the encrypted signal to the remote signal processing unit, the controller (Raspberry Pi **A**) is connected to a mobile phone (labeled **G**) that shares its Internet connection. The mobile phone also acts as a user interface to display the progression of the test. In our implementation we used a Google Android mobile phone LG Nexus 5 with a 4G connection. The Raspberry Pi and the Android device are connected through a micro-USB to USB cable. We developed an Android application that leverages the Android USB accessory API [27], which allows the phone to detect the Raspberry Pi as soon as it is connected and launches the corresponding app. This app has two purposes: it provides an interface for the user to start the blood test and provides a test progression feedback to the user via information on the screen, and relays the measurements to the cloud infrastructure, labeled **F**, in charge of performing the heavy computation. It also receives the analysis outcomes and forwards them to MEDSEN device. The Raspberry Pi runs a daemon listening for events on the USB port. When the phone is connected, the daemon exchanges information with the device using the Android Open Accessory Protocol [26]. This first exchange invites the user to download the diagnostic application from the Google Play Store. The implementation of the daemon running on the Raspberry Pi relies on `libusb` library via the `pyusb` package in order to detect events, read and write data on the USB ports. No specific security requirements for the user privacy are addressed at this layer. The mobile device is

not part of the trusted computing base and the valuable information confidentiality is already ensured though the encryption mechanisms above mentioned.

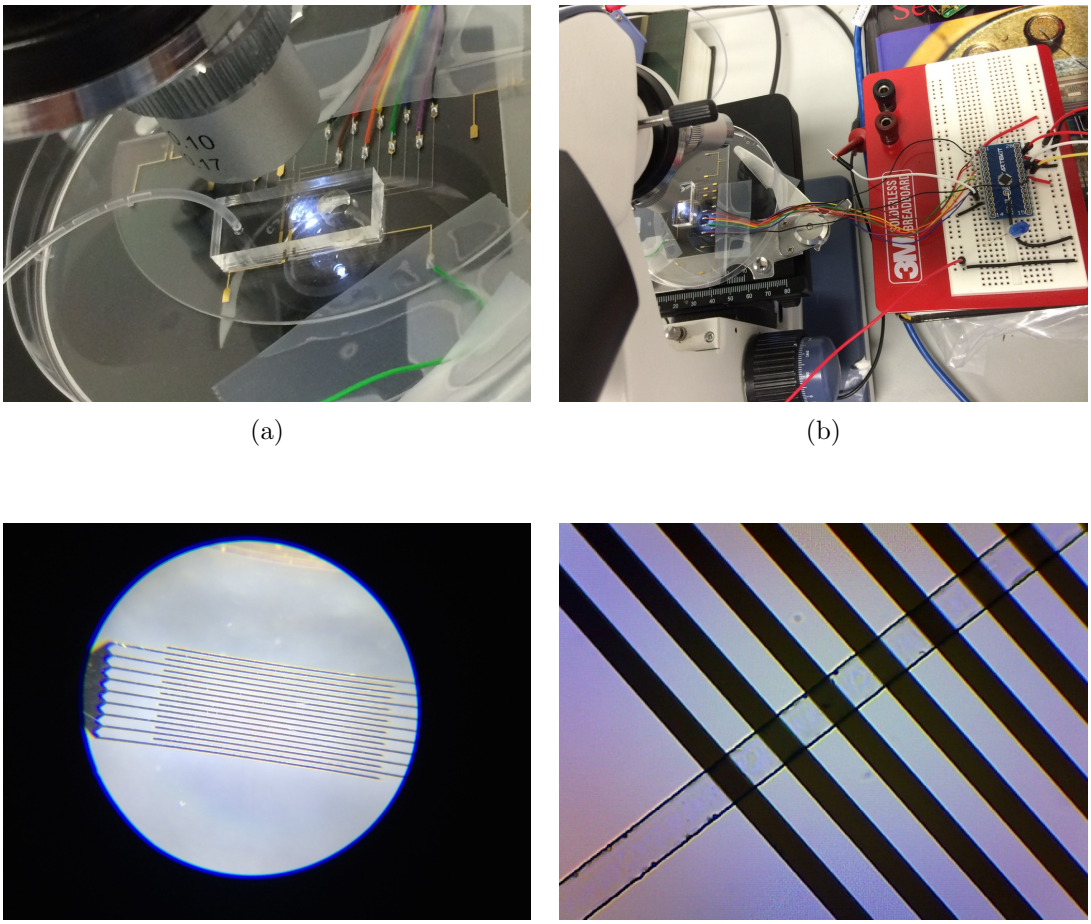
In our implementations for data acquisition, a Zurich Instruments HF2IS impedance spectroscope coupled with a HF2TA trans-impedance amplifier are used to measure the electrical impedance across the microfluidic channel. The bio-sensor in the microfluidic system is excited with the continual AC signals with a fixed discrete set of frequencies. The HF2IS impedance spectroscope can operate simultaneously at eight frequency carriers. The electrical impedance measurement between the electrode pairs in the microfluidic channel is modulated by the carrier frequencies. In recovering the signal measurement, the signal is demodulated by the same carrier frequencies. MEDSEN outputs the measurement from eight channels corresponding to the carrier frequencies, per measurement of electrical impedance. The choice of excitation voltage, frequencies, and measurement bandwidth is based on empirical test results of the system. The input electrode of the microfluidic channel is excited with a combination of $[500, 800, 1000, 1200, 1400, 2000, 3000, 4000]$kHz carrier frequencies. Excitation voltage is at 1 V per excitation signal. The recovered signal is sampled at 450 Hz. The recovering low pass filter is set to have cut off frequency at 120 Hz.

7 Evaluation

In our experiments, we evaluated the performance of the MEDSEN using micron-sized synthetic beads (synthetic beads 7.8 µm and 3.58 µm - MicroChem) as well as blood cells, suspended in PBS 0.9 %. The solution is pumped through the microfluidic channel at a rate of 0.08 µL.

7.1 Sensor-Based Data Encryption

Figure 4.11 illustrates how we can duplicate data generated for one electrode into multiple signals preventing the disclosure of number of beads passing through the channel. The figure shows the response of the bio-sensor to the 7.8 µm synthetic bead solution at 2 MHz. When selecting the random sequence of output electrodes, the remaining



(a) (b)

(c) (d)

Figure 4.10: Microfluidic sensor. (a) Microfluidic device under test. (b) Microfluidic device connected to multiplexer. (c) Image of fabricated biosensor. (d) Details of embedded electrodes in microfluidic channel (two parallel lines).

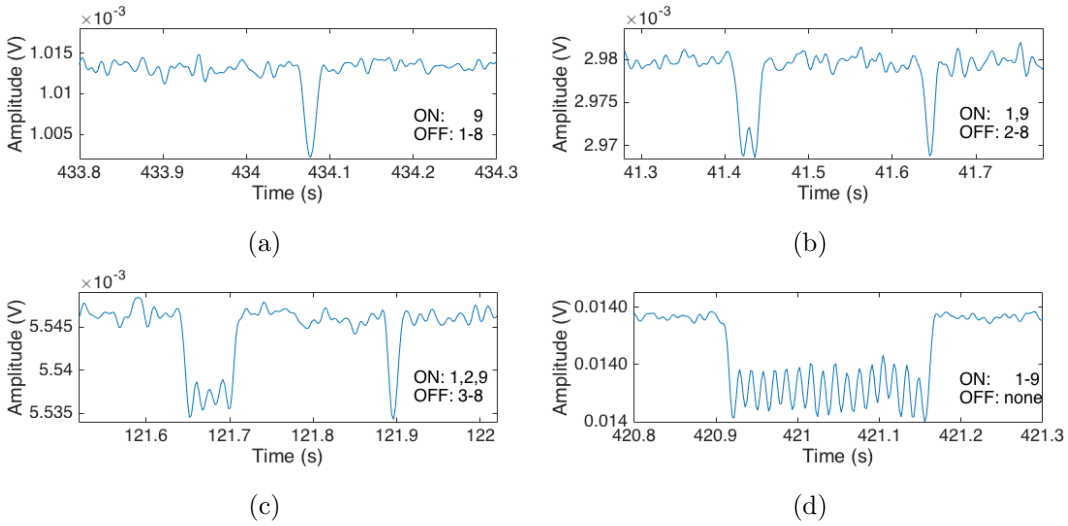


Figure 4.11: Representative encrypted cytometry data of a sensor with 9 input electrodes and 9 output electrodes detecting a single bead. Pseudo-random sequence selection of output electrodes. Output activated electrode numbers are specified. True number of peaks can only be detected/decrypted using unique key sequence.

unselected electrodes need to be grounded to prevent interference. Maxim Integrated MAX14661 16 : 2 multiplexer provides a dual output channel that can be utilized for this purpose. The encrypting algorithm will select a random sequence of output electrodes and route it to the first output channel of the multiplexer. The remaining unselected electrodes will be routed to the second output channel, which is proceeding to ground port. Figure 4.11a shows the measured response of the bio-sensor when one output electrode is selected and the remaining output electrodes are routed to the ground port. Figure 4.11b shows the response where the lead electrode (or electrode 9) is selected along with the last electrode (or electrode 1). Figure 4.11c shows the response of the bio-sensor when lead electrode 9 and electrode 1, 2 are selected. Figure 4.11d shows the outcomes when all the electrodes are activated. These measurements are then send for cloud-based peak detection analyses.

In Figure 4.11, the response time for each peak is approximately 20 ms. The distance each bead travels through a pair of electrodes, so a peak can be measured, is 45 µm (25 µm pitch, and 20 µm of two halves of electrode). The microfluidic channel dimension is 30 µm width, and 20 µm height. By dividing the volume of the solution passing through a pair of electrodes in the channel at the approximated time, the actual flow

rate in the channel can be calculated to be 0.081 µL/min.

MEDSEN's current deployment presents two limitations. First, the ninth electrode, for all signals (Figure 4.11), only generates one peak while all other electrodes generate double peaks. This is a minor fabrication flaw of the sensor that can be solved by adding another input electrode after the ninth electrode. Second, successive electrodes do not generate distinct non-differentiable peaks. Instead, a passing bead has an influence on multiple adjacent electrodes. Figure 4.11b illustrate this effect where the double peak at time 41.42s is not a double clone of the signal at time 41.65s. Similarly, if we consider multiple beads passing through the channel, we can notice that, due to the small distance interval between electrodes by comparison to the longer distance separating beads passing through the channel, there is a long delay between groups of peaks corresponding to a specific cell. This effect is illustrated in Figure 4.11d where all the electrodes are selected; the resulting signature is a relatively flat periodic train of 17 peaks, which is dissimilar from randomly passing cells. This information could be leveraged by a domain knowledgeable attacker to recover the true number of cells in the sample and thus the final diagnostic outcome. Both limitations can be solved by either putting more space between the electrodes or by selecting an electrode key pattern that does not use successive electrodes. Both of these changes are minor design modifications that increase the ciphertext strength against adversarial information disclosure attempts.

7.2 Data Transfer and Cloud-Based Analysis

To validate the accuracy of MEDSEN platform, we performed runtime diagnosis analysis multiple times over several blood samples. MEDSEN's typical diagnostics procedure takes a 0.01 mL of blood sample and complete all the steps, including sensor-side encryption, cloud processing, MEDSEN decoding and diagnostics, within 1 minute. However, to exercise and evaluate MEDSEN's ability to handle large data sets, we ran each sample through our bio-sensor for 3 h which generated approximately 600MB of encrypted biosensor measurements, captured in csv files. To improve the network transfer efficiency,

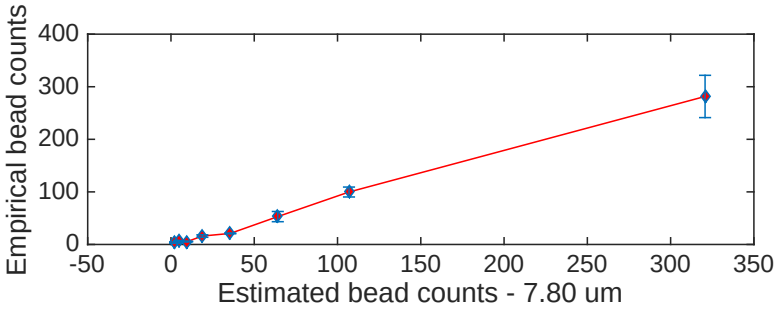


Figure 4.12: Measured bead counts vs number of beads expected for different concentrations of 7.8 µm synthetic beads.

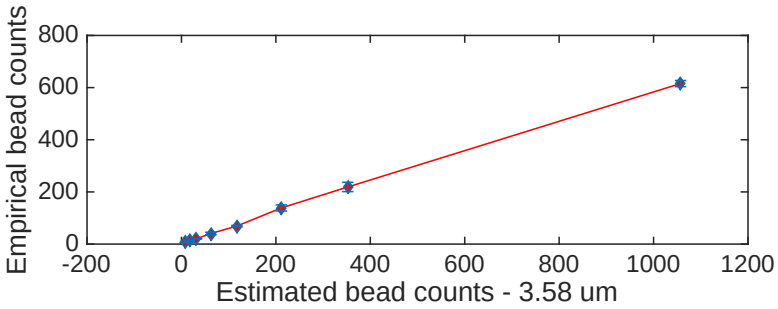


Figure 4.13: Measured bead count vs number of beads expected for different concentrations of 3.58 µm synthetic beads.

MEDSEN implements zip data compression on the smartphone. This reduced the sample size to 240MB. This provides a more adaptable solution to smartphone data plans when interacting with our cloud service. As discussed earlier in the chapter, the key size turns out to be less than 1 MB, i.e., 0.12MB accurately, that stays on the MEDSEN controller through the whole experiment. MEDSEN's design also allows (not implemented) sharing of the generated keys with trusted parties, e.g., the patient's practitioners, so that they could also access the cloud-based analysis outcomes remotely.

In peak-analysis, the accuracy of the bio-sensor is evaluated by comparing the empirically detected peaks and the estimated elements passing through the microfluidic channel. We diluted the 7.8 µm and 3.58 µm beads with PBS, which is a commonly used biological buffer that mimicks physiological samples like blood. We diluted at different concentrations to evaluate the empirical peak detection. The estimated number of elements in the solution is calculated according to the concentration information provided by the manufacturer, where we purchase the sample from. Four samples of each concentration are collected. The bead count data is taken from the first 5 min from each

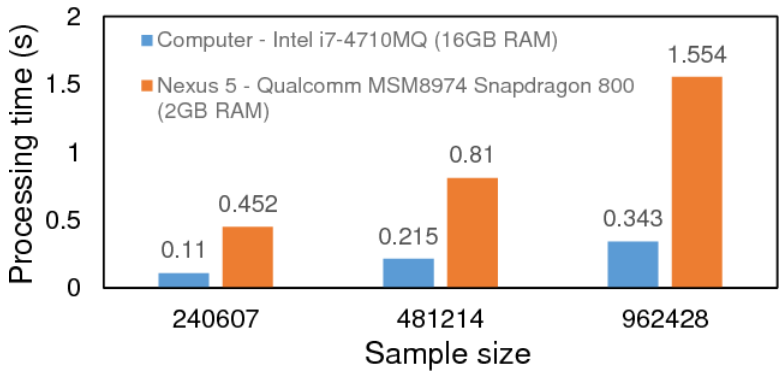
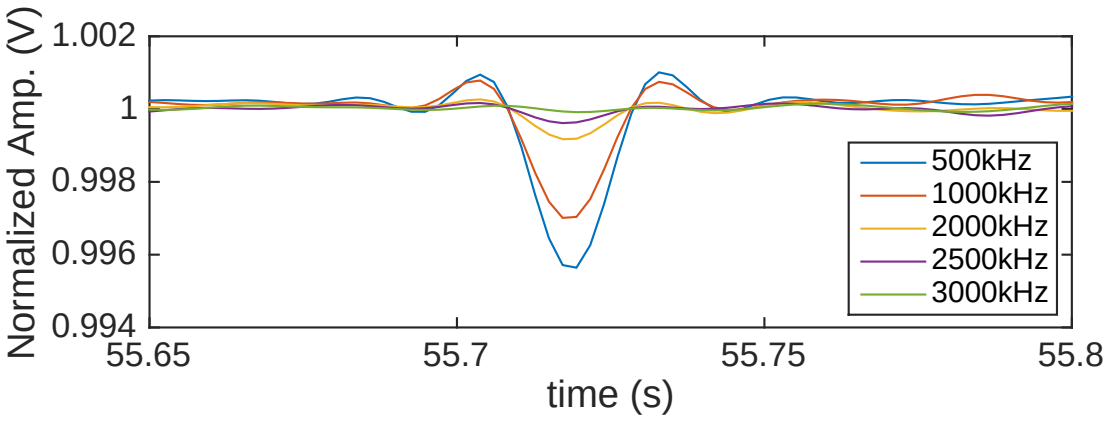


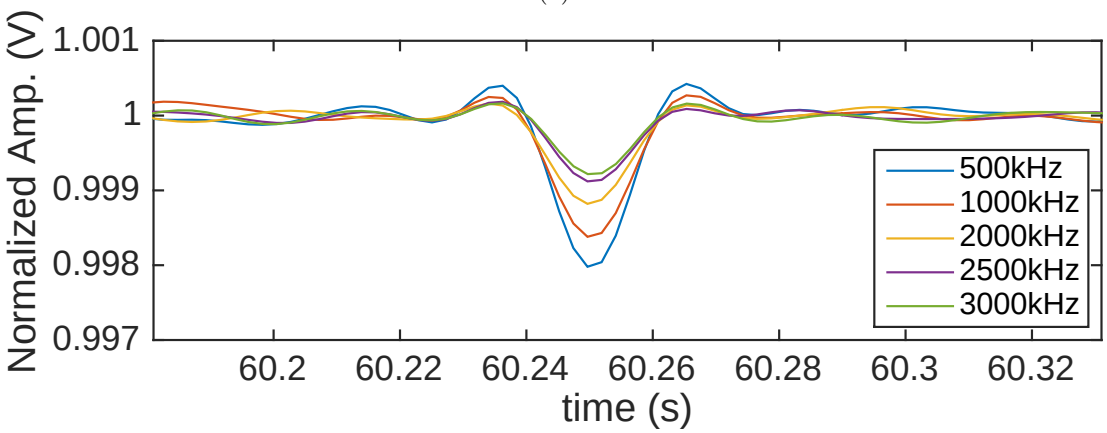
Figure 4.14: MEDSEN's Peak Analysis Performance on a Computer and Smartphone

sample. Figure 4.12 and 4.13 show the correlation of the empirical peak detection to the estimated peak counts in the microfluidic channel for 7.8 µm and 3.58 µm synthetic beads. As expected, the empirical peak detection varies linearly to the estimated peaks at different concentrations. The difference in bead counts is due to several reasons. For synthetic beads, the longer the experiments run, the more error would be expected on the empirical bead counts as many beads sink to the bottom of the inlet well and never make it to the sensor downstream in the micro-channel. The other reason for the bead count loss is due to the beads being adsorbed to microfluidic channel walls. These are issues that can be ultimately resolved with optimization of channel material and surface chemistry, which was beyond the scope of the current work.

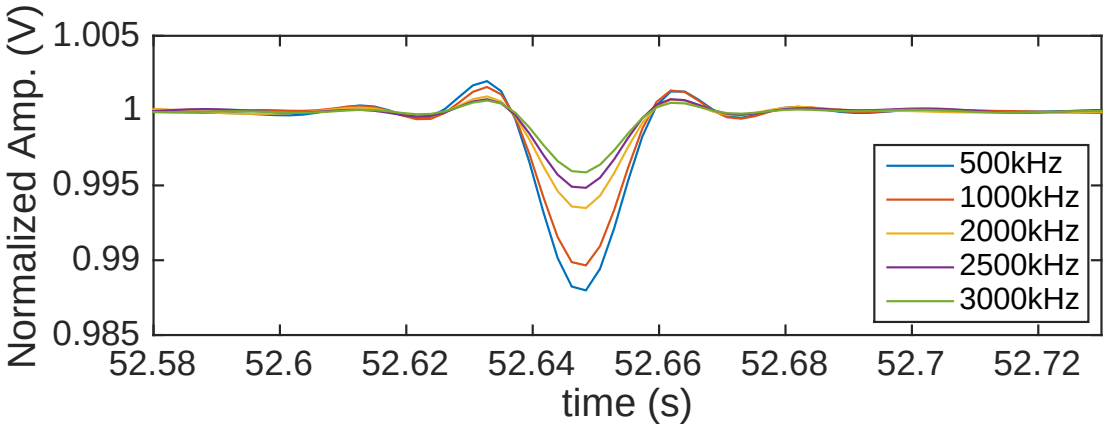
Figure 4.14 shows a performance comparison of the peak detection algorithm, when it runs on a standard computer system (possibly a cloud virtual machine) and on a smartphone device. It is noteworthy that a standard system provides much better performance than a mobile device, as the sample size grows larger. Aside from the storage capabilities, the enhanced computing power motivates the use of a cloud based service for handling peak detection and post-processing rather than using the smartphone. For smaller samples, however, MEDSEN could be configured to perform the peak counting signal processing on the smartphone locally.



(a)



(b)



(c)

Figure 4.15: Normalized impedance measurement of (a) blood cell, (b) 3.58 µm synthetic beads, and (c) 7.8 µm synthetic beads at different frequencies.

7.3 Cyto-Coded Passwords and Patient Authentication

Section 5 described MEDSEN's cyto-coded authentication using synthetic micro-beads mixed with the patient's blood sample. Every patient-specific unique identifier consists of a particular number (concentrations) of beads from different types. In conceptual comparison to traditional password paradigms, the number of password characters would correspond to the number of bead types involved, and specific character value within the password would correspond to the number (concentration) of beads of a particular type. Therefore, having larger number of bead types would increase the cyto-coded password space size and hence the overall security. The two crucial requirements, however, are *i)* MEDSEN's design and peak counting analysis can distinguish the peaks caused by the different types of beads; and *ii)* MEDSEN can distinguish different concentration levels of the same bead type within the blood samples for different patients. In other words, keeping concentration levels of two patients too close to each other may confuse MEDSEN, possibly lead to false user identification.

We evaluated the difference between the measured electrical impedance of 3.58 µm, 7.8 µm synthetic beads and actual blood cells. Figure 4.15 shows the results. The normalized electrical impedance of synthetic beads and blood cells is evaluated at multiple frequencies. Figure 4.15a shows at the frequency of 2 MHz and higher, the blood cell has lower electrical impedance response comparing to the impedance response of synthetic beads in Figure 4.15b and 4.15c. All those impedance measurements for different bead types at different frequencies are considered as features. MEDSEN uses the features for its classification procedures to distinguish between different particles. Figure 4.16 shows the results. The proposed solution is able to differentiate different types of synthetic beads and actual blood cells with clear margins. Furthermore, as discussed above, MEDSEN is able to recover the concentration of different types of beads quite precisely (Figures 4.12 and 4.13). Thus, MEDSEN can utilize both different bead types and bead concentration levels to generate unique identifiers in the patient blood sample efficiently. We noticed that low bead concentrations have less variance and improved resolution compared with higher concentrations (see the figures). This means that lower

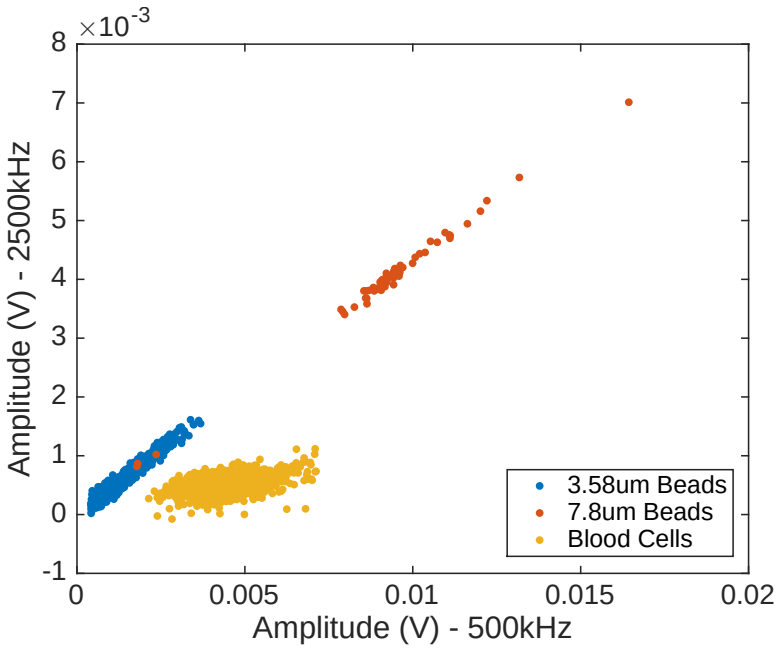


Figure 4.16: Cluster Representing Beads of Multiple Size for Password Generation

bead concentrations allow MEDSEN to define different concentration levels of the same bead types close to each other. This increases the password space size and entropy, and hence improves the design's overall security against bruteforce intrusions.

8 Related Work

Here, we review the most recent related past work that have been proposed in signal encryption, medical device security, and microfluidic biomarker detection.

General-purpose symmetric encryption [25] would require to decipher the samples on the server side for analysis and would reveal the clear dataset. Existing homomorphic encryption algorithms [44] currently do not provide the calculus flexibility and performance required to deal with biomarker sensor measurements. Additionally, conventional cryptography work on digital data points that would require addition of fairly complex analog-to-digital circuitry to MEDSEN's design.

For analog signal protection, the past work has proposed signal scrambling techniques [97] that implement a limited set of transformations using a key. In the medical

field, INTRAS proposes a key exchange and data encryption method based on interpolation and random sampling as an alternative symmetric encryption technique for electrocardiogram physiological signals [20]. These techniques are implemented in software, and require powerful processors to encrypt fine-grained analog signals once original measurements are acquired by the sensor hardware. MEDSEN reconfigures the sensor hardware such that the acquired measurements are already encrypted. This eliminates the need for powerful computational and memory resources as large trusted computing bases. Hence it brings down the size, complexity and cost of the device, while improving the overall security. To the best of our knowledge, MEDSEN is the first physical based encryption scheme for cytometry that do not rely on any software-based analog or digital signal manipulation.

Flow cytometry has been studied extensively as alternative methods to impedance cytometry for diagnosing and monitoring diseases such as HIV, malaria, and tuberculosis [13, 65]. For instance, [13] has shown the high correlation of CD4+T-lymphocytes counts by flow cytometry and the standard of Coulter cytosphere assay. White blood cell counts have also been studied to characterize Plasmodium falciparum-infected patients, Plasmodium vivax-infected patients, and the uninfected patients [65]. However, the technique is expensive and requires highly trained technicians adhering to the strict protocols. These challenges call for the design and development of cost-effective disposable testing solutions without sacrificing the sensitivity[33]. Microfluidic protein quantification also has been conducted using a mobile platform [37, 63, 59]. The protein is aggregated with gold nanoparticles and detected with LED light . The results from the experiment are stored in text file and distributed over the network via Google Drive. However, for sensitive medical information, such as HIV diagnosis, the results should be kept secured for patient's privacy. Our method now enables for higher diagnostic accuracy through single-cell and single particle detection, but also for embedding security at the physical sensor level.

9 Conclusion

In this chapter, we presented MEDSEN, a portable point-of-care disease diagnostics solutions that ensures low-cost and accurate outcomes through use of the smartphone computational resources. MEDSEN provides in-sensor hardware-based analog signal encryption along with cyto-coded authentication services. MEDSEN's specific encryption design enables cloud-based analysis of encrypted analog signals without disclosing the users' privacy and confidential medical information. Our real-world implementation of MEDSEN's bio-sensor circuitry and software stack proves its accurate and secure diagnostics capabilities empirically.

Chapter 5

Conclusion

The smartphones have brought a lot of convenience to perform daily tasks. Their wide adoption, their proximity to the end users and the large range of sensors they provide make them a target of choice to gather and access detailed customer data. Protecting this data protection is a priority for end users as well as for corporations providing services through these devices. This thesis detailed three complementary approaches to solve this problem.

Contributions of this thesis

- Chapter 2 introduces the concept of virtual micro-security perimeters to provide a dynamic separation between data from different sources without the cost of a full virtualization. Our approach utilizes an hybrid information flow tracking system that is able to provide two level of data tracking: a very fine grained variable-level data tracking for application that do support it, or a application-level data tracking approach for applications that do not support it. We propose a prototype of the solution on top of commodity devices and evaluate our approach though the use of real-word third-party apps.

- Chapter 3 proposes a new approach to track numerical information flows. By limiting the focus on numerical type, we demonstrate how to achieve a good coverage of information flow with a small instrumentation requirement by only monitoring numerical operations. We detail an implementation of this solution via a third-party application, that do not require any system changes on the target device.

- Chapter 4 considers a data protection mechanism for point-of-care medical devices with a very limited trusted computing base. The solution provides an analog signal encryption scheme that operates on a cytometry sensor. This domain specific scrambling scheme masks the exact measurements, while the ciphertext still conserves enough information such that the analysis can be done on a curious but honest system without the risk of revealing the diagnostic outcome. Chapter 4 also presents an authentication scheme that relies on enhancing the measured test sample to create a transparent authentication scheme.

Future lines of approach

Tailoring the use of information flow tracking to specific use cases. As seen in Chapter 2 and Chapter 3, the use of information flow tracking system has limits based on the type of use cases, applications or implementations that are considered. In particular, an approach based on numerical values such as the one presented in Metron would greatly benefit embedded devices that only manipulate numerical values such as Internet of Things devices or cyber-physical systems.

The use of hardware facilities to complement information flow tracking system. Recent advances in processor technologies such as ARM TrustZone and Intel SGX would greatly benefit the models presented in Chapter 2 and Chapter 3. Mainly, the storage of policies and flow information aside from the target system or program memory would greatly improve the threat model for our approaches.

Binary rewriting approaches for information flow tracking systems. Chapter 2 and Chapter 3 require changes on the target application and on the compiler respectively to implement information flow tracking for the target application. In the first case, this require extensive system changes and application changes in order to support some features while in the second case, it requires recompiling the source code. Many applications are not provide with their source code. In this case, the possibility to rewrite and instrument a target binary would allow to automatize the use of

information flow tracking for general COTS applications. This instrumentation effort would benefit from a prior simplification of the binary, as the complexity of the software implementation usually generates

Bibliography

[1] 2018, Android private data leak benchmark available at https://github.com/secure-software-engineering/DroidBench/blob/master/eclipse-project/AndroidSpecific/PrivateDataLeak3/src/de/ecspride/MainActivity.java.

[2] 2018, Android Implicit flow benchmark available at https://github.com/secure-software-engineering/DroidBench/blob/master/eclipse-project/ImplicitFlows/ImplicitFlow1/src/de/ecspride/ImplicitFlow1.java.

[3] 2018, Android Loop1 benchmark available at https://github.com/secure-software-engineering/DroidBench/blob/master/eclipse-project/GeneralJava/Loop1/src/de/ecspride/LoopExample1.java.

[4] "System call overhead," http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf.

[5] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[6] Android, 2015, location API; available at https://developer.android.com/training/location.

[7] Android Developers Manual - Customizing SELinux, 2015, https://source.android.com/devices/tech/security/selinux/customize.html.

[8] J. Andrus, C. Dall, A. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 173–187.

[9] AnTuTu, 2015, android AnTuTu performance benchmark; available at http://www.antutu.net.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[11] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 691–706.

[12] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," *CoRR*, vol. abs/1607.06619, 2016. [Online]. Available: http://arxiv.org/abs/1607.06619

[13] P. Balakrishnan, M. Dunne, N. Kumarasamy, S. Crowe, G. Subbulakshmi, A. K. Ganesh, A. J. Cecelia, P. Roth, K. H. Mayer, S. P. Thyagarajan, and S. Solomon, "An inexpensive, simple, and manual method of cd4 t-cell quantitation in hiv-infected individuals for use in developing countries," *JAIDS Journal of Acquired Immune Deficiency Syndromes*, vol. 36, no. 5, pp. 1006–1010, 2004.

[14] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The vmware mobile virtualization platform: is that a hypervisor in your pocket?" *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 124–135, 2010.

[15] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 27–38.

[16] Bluebox, 2014, https://bluebox.com/.

[17] J. A. Bonachela, H. Hinrichsen, and M. A. Munoz, "Entropy estimates of small data sets," *Journal of Physics A: Mathematical and Theoretical*, vol. 41, no. 20, p. 202001, 2008.

[18] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 51–62.

[19] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies." in *Usenix security*, 2013, pp. 131–146.

[20] F. M. Bui and D. Hatzinakos, "Biometric methods for secure communications in body sensor networks: Resource-efficient key management and signal-level data scrambling," *EURASIP J. Adv. Signal Process*, vol. 2008, pp. 109:1–109:16, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1155/2008/529879

[21] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipshield: a framework for enforcing context-aware privacy," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 143–156.

[22] K. Z. Chen, N. Johnson, V. DSilva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual policy enforcement in android applications with permission event graphs," in *Proc. NDSS*, 2013.

[23] X. Cheng, D. Irimia, M. Dixon, J. C. Ziperstein, U. Demirci, L. Zamir, R. G. Tompkins, M. Toner, and W. R. Rodriguez, "A microchip approach for practical label-free cd4+ t-cell counting of hiv-infected subjects in resource-poor settings," *JAIDS Journal of Acquired Immune Deficiency Syndromes*, vol. 45, no. 3, pp. 257–261, 2007.

[24] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime."

[25] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[26] A. Developers, "Android Open Accessory Protocol," https://source.android.com/accessories/protocol.html, 2015, [Online; accessed 19-July-2015].

[27] ——, "USB Accessory," https://developer.android.com/guide/topics/connectivity/usb/accessory.html, 2015, [Online; accessed 19-July-2015].

[28] ——, 2017, android Motion Sensors; available at https://developer.android.com/guide/topics/sensors/sensors_motion.html.

[29] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, 2011.

[30] D. C. Duffy, J. C. McDonald, O. J. Schueller, and G. M. Whitesides, "Rapid prototyping of microfluidic systems in poly (dimethylsiloxane)," *Analytical chemistry*, vol. 70, no. 23, pp. 4974–4984, 1998.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 17–30, 2005.

[32] J. M. Eisenberg, "Can you keep a secret?" *Journal of general internal medicine*, vol. 16, no. 2, pp. 131–133, 2001.

[33] S. Emaminejad, M. Javanmard, R. W. Dutton, and R. W. Davis, "Microfluidic diagnostic tool for the developing world: Contactless impedance flow cytometry," *Lab on a Chip*, vol. 12, no. 21, pp. 4499–4507, 2012.

[34] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security & Privacy, IEEE*, vol. 7, no. 1, pp. 50–57, 2009.

[35] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–6.

[36] N. Engel and N. Pant Pai, "Qualitative research on point-of-care testing strategies and programs for hiv," *Expert review of molecular diagnostics*, no. 0, pp. 1–5, 2015.

[37] D. Erickson, D. O'Dell, L. Jiang, V. Oncescu, A. Gumus, S. Lee, M. Mancuso, and S. Mehta, "Smartphone technology can be transformative to the deployment of lab-on-chip diagnostics," *Lab on a Chip*, vol. 14, no. 17, pp. 3159–3164, 2014.

[38] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab'n run: Secure and practical dynamic code loading for android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 201–210.

[39] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[40] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association, 2011, pp. 7–7.

[41] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[42] I. for Health Freedom, "Public attitudes toward medical privacy. report submitted by the gallup organization; available at http://www.forhealthfreedom.org/Gallupsurvey/," 2001.

[43] E. Garcia-Ceja, V. Osmani, and O. Mayora, "Automatic stress detection in working environments from smartphones accelerometer data: a first step," *IEEE journal of biomedical and health informatics*, vol. 20, no. 4, pp. 1053–1060, 2016.

[44] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *STOC*, vol. 9, 2009, pp. 169–178.

[45] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[46] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[47] B. Greve, R. Kelsch, K. Spaniol, H. T. Eich, and M. Götte, "Flow cytometry in cancer stem cell analysis and separation," *Cytometry Part A*, vol. 81, no. 4, pp. 284–293, 2012.

[48] V. Gubala, L. F. Harris, A. J. Ricco, M. X. Tan, and D. E. Williams, "Point of care diagnostics: status and future," *Analytical chemistry*, vol. 84, no. 2, pp. 487–515, 2011.

[49] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: a programmable interface for extending android security," *Intel CRI-SC at TU Darmstadt, North Carolina State University, CASED/TU Darmstadt, Tech. Rep. TUD-CS-2014-0063*, 2014.

[50] H. Hillborg and U. Gedde, "Hydrophobicity changes in silicone rubbers," *IEEE Transactions on Dielectrics and Electrical insulation*, vol. 6, no. 5, pp. 703–717, 1999.

[51] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 639–652.

[52] Invisible Things Lab, 2011, http://www.qubes-os.org/.

[53] A. Jayakumar, "Cyberattacks are on the rise, and healthcare data is the biggest target; available at http://www.washingtonpost.com/," 2014.

[54] B.-H. Jo, L. M. Van Lerberghe, K. M. Motsegood, and D. J. Beebe, "Three-dimensional micro-channel fabrication in polydimethylsiloxane (pdms) elastomer," *Microelectromechanical Systems, Journal of*, vol. 9, no. 1, pp. 76–81, 2000.

[55] W. Jung, J. Han, J.-W. Choi, and C. H. Ahn, "Point-of-care testing (poct) diagnostic systems using microfluidic lab-on-a-chip technologies," *Microelectronic Engineering*, vol. 132, pp. 46–57, 2015.

[56] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 321–334.

[57] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: a generic operating system framework for secure smartphones," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 39–50.

[58] O. Lazcka, F. Campo, and F. X. Munoz, "Pathogen detection: A perspective of traditional methods and biosensors," *Biosensors and Bioelectronics*, vol. 22, no. 7, pp. 1205–1217, 2007.

[59] S. Lee, V. Oncescu, M. Mancuso, S. Mehta, and D. Erickson, "A smartphone platform for the quantification of vitamin d levels," *Lab on a Chip*, vol. 14, no. 8, pp. 1437–1442, 2014.

[60] C. LeRouge, V. Mantzana, and E. V. Wilson, "Healthcare information systems research, revelations and visions," *European Journal of Information Systems*, vol. 16, no. 6, p. 669, 2007.

[61] X. Liu, T.-Y. Lin, and P. B. Lillehoj, "Smartphones for cell and biomolecular detection," *Annals of biomedical engineering*, vol. 42, no. 11, pp. 2205–2217, 2014.

[62] C. Logan, M. Givens, J. T. Ives, M. Delaney, M. J. Lochhead, R. T. Schooley, and C. A. Benson, "Performance evaluation of the mbio diagnostics point-of-care cd4 counter," *Journal of immunological methods*, vol. 387, no. 1, pp. 107–113, 2013.

[63] M. Mancuso, E. Cesarman, and D. Erickson, "Detection of kaposi's sarcoma associated herpesvirus nucleic acids using a smartphone accessory," *Lab on a Chip*, vol. 14, no. 19, pp. 3809–3816, 2014.

[64] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings*

of the 18th ACM conference on Computer and communications security. ACM, 2011, pp. 551–562.

[65] F. E. McKenzie, W. A. Prudhomme, A. J. Magill, J. R. Forney, B. Permpanich, C. Lucas, R. A. Gasser, and C. Wongsrichanalai, "White blood cell counts and malaria," *Journal of Infectious Diseases*, vol. 192, no. 2, pp. 323–330, 2005.

[66] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Proceedings of the 10th international conference on Mobile systems, applications, and services.* ACM, 2012, pp. 323–336.

[67] J. Mok, M. N. Mindrinos, R. W. Davis, and M. Javanmard, "Digital microfluidic assay for protein detection," *Proceedings of the National Academy of Sciences*, vol. 111, no. 6, pp. 2110–2115, 2014.

[68] A. Nadkarni and W. Enck, "Preventing accidental data disclosure in modern operating systems," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1029–1042. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516677

[69] M. Ongtang, K. Butler, and P. McDaniel, "Porscha: Policy oriented secure content handling in android," in *Proceedings of the 26th Annual Computer Security Applications Conference.* ACM, 2010, pp. 221–230.

[70] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," *Security and Communication Networks*, vol. 5, no. 6, pp. 658–673, 2012.

[71] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok, "Unionfs: User-and community-oriented development of a unification filesystem," in *Proceedings of the 2006 Linux Symposium*, vol. 2, 2006, pp. 349–362.

[72] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, 2014.

[73] R. A. Rueppel, "Stream ciphers," in *Analysis and Design of Stream Ciphers.* Springer, 1986, pp. 5–16.

[74] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Commun. ACM*, vol. 58, no. 4, pp. 74–82, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2667218

[75] Samsung Knox, 2014, http://www.samsungknox.com/.

[76] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2295136.2295141

[77] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proceedings of the*

17th ACM symposium on Access Control Models and Technologies. ACM, 2012, pp. 13–22.

[78] SensMark, 2018, the Benchmark For Mobile Sensor Technologies; available at http://sensmark.info.

[79] P. Shekelle, S. C. Morton, and E. B. Keeler, "Costs and benefits of health information technology," 2006.

[80] S. Smalley, 2015, sE-Android; available at http://seandroid.bitbucket.org/.

[81] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*. IEEE, 2013.

[82] R. Spreitzer, "Pin skimming: Exploiting the ambient-light sensor in mobile devices," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014, pp. 51–62.

[83] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.

[84] Swirls, 2015, swirls anonymous demonstration; available at http://goo.gl/ofsC5N (capsule installation) and http://tinyurl.com/knvg77a (capsule boundary tracking and policy enforcement).

[85] S. H. Tan, N.-T. Nguyen, Y. C. Chua, and T. G. Kang, "Oxygen plasma treatment for reducing hydrophobicity of a sealed polydimethylsiloxane microchannel," *Biomicrofluidics*, vol. 4, no. 3, p. 032204, 2010.

[86] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction." in *OSDI*, vol. 12, 2012, pp. 77–91.

[87] O. Tripp and J. Rubin, "A bayesian approach to privacy enforcement in smartphones," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 175–190.

[88] V. Velusamy, K. Arshak, O. Korostynska, K. Oliwa, and C. Adley, "An overview of foodborne pathogen detection: in the perspective of biosensors," *Biotechnology advances*, vol. 28, no. 2, pp. 232–254, 2010.

[89] R. S. Wallis, M. Pai, D. Menzies, T. M. Doherty, G. Walzl, M. D. Perkins, and A. Zumla, "Biomarkers and diagnostics for tuberculosis: progress, needs, and translation into practice," *The Lancet*, vol. 375, no. 9729, pp. 1920–1937, 2010.

[90] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab, "Easeandroid: automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning," in *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, 2015, pp. 351–366.

[91] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices." in *NDSS*, 2015.

[92] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0." in *Usenix Security*, vol. 1999, 1999.

[93] Y. Xia and G. M. Whitesides, "Soft lithography," *Annual review of materials science*, vol. 28, no. 1, pp. 153–184, 1998.

[94] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *USENIX conference on Security symposium*, 2012, pp. 29–29.

[95] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," pp. 85–96, 2012.

[96] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, vol. 7, 2006, pp. 19–19.

[97] W. Zeng and S. Lei, "Efficient frequency domain selective scrambling of digital video," *Multimedia, IEEE Transactions on*, vol. 5, no. 1, pp. 118–129, 2003.