Elastic Phone

Towards Detecting and Mitigating Computation and Energy Inefficiencies in Mobile Apps

Mario Manuel Sa Dias e Pinto de Almeida

Supervisor: Jeremy Blackburn

Tutor: Leandro Navarro Moldes

Department of Computer Architecture

Polytechnic University of Catalonia

This dissertation is submitted for the degree of

Doctor of Computer Science

May 2018

“O pensamento pode ter elevação sem ter elegância, e, na proporção em que não tiver elegância, perderá a ação sobre os outros. A força sem a destreza é uma simples massa.”

- Livro do Desassossego, Bernardo Soares (Fernando Pessoa’s heteronym).

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Mario Manuel Sa Dias e Pinto de Almeida
May 2018

Acknowledgements

A warm thanks to my family, perhaps the greatest influence in my pursue for knowledge. To Andreia, for her love and endurance during all the moments I doubted my professional choices. To Dina and Yan for believing and welcoming me to Telefonica. To Jeremy for his guidance throughout this thesis and for helping relief some stress online. To the Telefonica Research family with whom i shared so many moments. To Leandro for his awesome EMDC program and for being present throughout some of my best years in life. To John, Liang and all the great people in Cambridge a big thank you for the hospitality. To the EMDC mafia: Anis, Bilal, Ioanna, J. Neto, Leonardo, Manos, Marcus, Maria, Navaneeth, Sana... too many to name, sorry, I love you all! To all my friends in Barcelona whom made my stay enjoyable, with special mentions to Alex, Daniela and Ilias. Finally, to all the others i did not mention but shaped my life in any given way.

Abstract

Mobile devices have become ubiquitous and their ever evolving capabilities are bringing them closer to personal computers. Nonetheless, due to their mobility and small size factor constraints, they still present many hardware and software challenges. Their limited battery life time has led to the design of mobile networks that are inherently different from previous networks (e.g., wifi) and more restrictive task scheduling. Additionally, mobile device ecosystems are more susceptible to the heterogeneity of hardware and from conflicting interests of distributors, internet service providers, manufacturers, developers, etc.

The high number of stakeholders ultimately responsible for the performance of a device, results in an inconsistent behavior and makes it very challenging to build a solution that improves resource usage in most cases.

The focus of this thesis is on the study and development of techniques to detect and mitigate computation and energy inefficiencies in mobile apps. It follows a bottom-up approach, starting from the challenges behind detecting inefficient execution scheduling by looking only at apps' implementations. It shows that scheduling APIs are largely misused and have a great impact on devices wake up frequency and on the efficiency of existing energy saving techniques (e.g., batching scheduled executions).

Then it addresses many challenges of app testing in the dynamic analysis field. More specifically, how to scale mobile app testing with realistic user input and how to analyze closed source apps' code at runtime, showing that introducing humans in the app testing loop improves the coverage of app's code and generated network volume.

Finally, using the combined knowledge of static and dynamic analysis, it focuses on the challenges of identifying the resource hungry sections of apps and how to improve their execution via offloading. There is a special focus on performing non-intrusive offloading transparent to existing apps and on in-network computation offloading and distribution. It shows that, even without a custom OS or app modifications, in-network offloading is still possible, greatly improving execution times, energy consumption and reducing both end-user experienced latency and request drop rates. It concludes with a real app measurement study, showing that a good portion of the most popular apps' code can indeed be offloaded and proposes future directions for the app testing and computation offloading fields.

Resumen

Los dispositivos móviles se han tornado omnipresentes y sus capacidades están en constante evolución acercándolos a los computadoras personales. Sin embargo, debido a su movilidad y tamaño reducido, todavía presentan muchos desafíos de hardware y software. Su duración limitada de batería ha llevado al diseño de redes móviles que son inherentemente diferentes de las redes anteriores y una programación de tareas más restrictiva. Además, los ecosistemas de dispositivos móviles son más susceptibles a la heterogeneidad de hardware y los intereses conflictivos de las entidades responsables por el rendimiento final de un dispositivo.

El objetivo de esta tesis es el estudio y desarrollo de técnicas para detectar y mitigar las ineficiencias de computación y energéticas en las aplicaciones móviles. Empieza con los desafíos detrás de la detección de planificación de ejecución ineficientes, mirando sólo la implementación de las aplicaciones. Se muestra que las API de planificación son en gran medida mal utilizadas y tienen un gran impacto en la frecuencia con que los dispositivos despiertan y en la eficiencia de las técnicas de ahorro de energía existentes.

A continuación, aborda muchos desafíos de las pruebas de aplicaciones en el campo de análisis dinámica. Más específicamente, cómo escalar las pruebas de aplicaciones móviles con una interacción realista y cómo analizar código de aplicaciones de código cerrado durante la ejecución, mostrando que la introducción de humanos en el bucle de prueba de aplicaciones mejora la cobertura del código y el volumen de comunicación de red generado.

Por último, combinando la análisis estática y dinámica, se centra en los desafíos de identificar las secciones de aplicaciones con uso intensivo de recursos y cómo mejorar su ejecución a través de la ejecución remota (i.e., “offload”). Hay un enfoque especial en el “offload” no intrusivo y transparente a las aplicaciones existentes y en el “offload” y distribución de computación dentro de la red. Demuestra que, incluso sin un sistema operativo personalizado o modificaciones en la aplicación, el “offload” en red sigue siendo posible, mejorando los tiempos de ejecución, el consumo de energía y reduciendo la latencia del usuario final y las tasas de caída de solicitudes de “offload”. Concluye con un estudio real de las aplicaciones más populares, mostrando que una buena parte de su código puede de hecho ser ejecutado remotamente y propone direcciones futuras para los campos de “offload” de aplicaciones.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Mobile devices have become ubiquitous and their ever evolving capabilities are bringing them closer to personal computers. Nonetheless, due to their mobility and small size factor constraints, they still present many hardware and software challenges. Their limited battery life time has led to the design of mobile networks that are inherently different from previous networks (e.g., wifi) and more restrictive task scheduling. Additionally, mobile device ecosystems are more susceptible to the heterogeneity of hardware and from conflicting interests of distributors, internet service providers, manufacturers, developers, etc. The high number of stakeholders ultimately responsible for the performance of a device, results in an inconsistent behavior and makes it very challenging to build a solution that improves resource usage in most cases.

The focus of this thesis is on the study and development of techniques to detect and mitigate computation and energy inefficiencies in mobile apps. Detecting inefficiencies can be achieved by studying app's code (e.g., detecting locks that prevent the device from sleeping [99]) or by analyzing the app during execution (e.g., measuring app's energy consumption during runtime [160]). This thesis follows a bottom-up approach to mitigate computation and energy inefficiencies. First, it studies apps' implementations (via static analysis) and focuses on a functionality that is core to Android's energy saving mechanisms, i.e., alarms, a set of APIs that handle execution scheduling.

Second, it addresses some of the challenges in testing and analyzing apps during runtime (via dynamic analysis), more specifically, the struggle faced by researchers when trying to scale mobile app testing with *realistic user input*. It does so by integrating with crowdsourcing platforms and streaming a virtualized mobile OS (Android), while performing runtime analysis of the executed app's code. Unlike previous literature [16, 18, 39, 40, 100], it's runtime analysis allows the retrieval of information such as code coverage and similarity between runs, even for closed-source apps.

Finally, using the combined knowledge of static and dynamic analysis, this thesis targets the challenges of identifying the resource hungry sections of apps and how to improve their execution. While there are many techniques available for improving the energy consumption of apps, this thesis focuses on computation offloading (i.e., executing computation remotely) which was shown to be extremely effective [45], but still comes with a few limitations. More specifically, it aims to provide non-intrusive offloading, i.e., without custom OSes or app modifications, and improved computation distribution (i.e., executing closer to the user), complementing the evaluation of the proposed solution with the analysis of real apps to determine their offloading feasibility.

1.1 Execution Scheduling

First, this thesis tackles the challenges in studying **what** apps can do from the perspective of execution scheduling, analyzing apps' structure and impact of developer decisions and OS fragmentation in the app resource usage efficiency.

Todays mobile devices support a diverse set of functionality, much of which is not dependent on active user interaction. Many tasks are performed in the background, which has very clear impact on battery life and mobile data usage [29]. *Alarms* are Android's integrated application execution scheduling mechanism (used, e.g., for background network activity) and are a primary vehicle for improving devices energy efficiency. One way Android mitigates Alarms' negative impact is *batching*, i.e., by executing alarms with (reasonably) proximate execution scheduled times together. Batching can greatly reduce total device awake time while increasing the chance that traffic from different applications can occur simultaneously (and thus greatly reducing energy consumption [26, 33, 79, 98, 112, 149]). Unfortunately, the success of batching depends on the correct usage of alarm APIs (by applications) which widespread and impact are unknown.

In this thesis (Chapter 4), a large scale study of over 22 thousand of the most popular applications in the Google Play Market is performed to quantify whether expectations of more energy efficient background app execution are indeed warranted. It attempts to determine if there is a chasm between the way application developers build their apps and Android's attempt to address energy inefficiencies of background app execution. To do so, it shows that many of the app execution scheduling (alarms) misuses can be detected by statically analyzing app's code (e.g., by analyzing API adoption), and demonstrates their impact on the device performance at runtime.

It concludes with a discussion on how the current scheduling inefficiencies could be addressed and stresses that research on energy efficiency on mobile devices needs to incorporate an understanding around the use of alarms.

1.2 Large Scale Dynamic Analysis

The thesis then delves into the challenges behind analyzing **how** apps execute and how to test them with realistic human-like inputs.

While static analysis is widely used in security [104] and computation offloading research [161], it studies **what** an application can do but fails to capture app's runtime behavior. Dynamic properties such as the app's dynamic execution control flow (**how** the app executes) and invocation counts are crucial for understanding the resource requirements of apps. Unfortunately, realistic app testing is still challenging at scale with existing solutions often relying on distributing specially instrumented apps, or even phones with pre-installed apps, to real users. This approach is quite expensive and has intrinsic scalability limitations.

Therefore, in this thesis (Chapter 5), a solution for enabling large-scale app testing and runtime tracing is proposed and evaluated, along with the key design principles that allow researchers to collect human inputs, priorly difficult to collect at scale. This is important for improving automation characteristics (e.g., improve the coverage of tested code) and for building new automation tools that learn from humans and perform more realistic interactions (e.g., for avoiding automation detection and evasion techniques by malware apps [102]).

The proposed solution integrates existing crowdsourcing platforms for acquiring users, while distributing the apps via streaming to users' web browsers. Unlike previous UI automation literature [16, 18, 39, 40, 100] which relied on simple and unrealistic open-source apps, in this study, a new VM profiling mechanism is introduced that enables the extraction of popular UI automation benchmarking metrics (i.e., code coverage and similarity) to be collected even for popular closed-source apps. Using these metrics, a comparison between humans and the defacto UI automation tool (i.e., the `monkey`) performances is provided, showing that introducing humans in the test loop does indeed improve the coverage of apps.

Finally, a promising future direction is proposed, where a RNN-based prototype trained with human interactions is used to improve over the existing UI automation tools, reducing the number of clicks and time required to achieve common tasks.

1.3 Dynamic Code Execution and Offloading

Finally, using the combined knowledge of static and dynamic analysis, this thesis targets the challenges of detecting **when** code execution can be optimized in terms of energy consumption.

Recent work has proposed various solutions to detect, offload and execute computationally intensive functionality of mobile apps remotely in a cloud, referred to as a mobile-to-cloud paradigm [38, 41, 45, 73, 89, 93, 161]. Their evaluations have shown that the energy consumption of CPU intensive apps can be reduced by an order of magnitude [41, 45]. Unfortunately, these works either failed to address the challenges of deploying and scaling mobile code offloading systems at all, or overlooked the opportunities to effectively exploit in-network resources. Furthermore, their solutions utilize intrusive offloading techniques which either require custom OS distributions, app repackaging, or even alternative app stores, which not only increases security risks and deployment costs, but also greatly increases the barrier to the market adoption.

Therefore, in Chapter 6, a solution for in-network offloading of mobile app computation is proposed and evaluated, along with the key design principles that allow non-intrusive offloading. By targeting a single Android binary (e.g., via a fully reversible binary patch) from which all apps' processes are forked, it demonstrates how apps can be extended to provide offloading, improving their execution time and energy consumption without relying on a custom OS or app modifications. Unlike previous solutions, this work has a special focus on in-network computation offloading and distribution, it proposes and compares different strategies to effectively balance functionality load while reducing both end-user-experienced latency and request drop rates.

It concludes with two real app measurement studies of the most popular Google Play apps, showing a great potential for reducing the costs of hosting apps, and quantify the amount of these apps' code that can indeed be offloaded.

1.4 Thesis Overview

All together, the three topics aforementioned provide solutions to detect and mitigate computation and energy inefficiencies in mobile apps.

Chapter 2 provides the background and state of art for the covered research topics. Then Chapter 3 states the research problems and contributions of the thesis.

The thesis then delves into each of the research topics. First, Chapter 4 describes the study of Android's execution scheduling. Second, Chapter 5 presents the large scale dynamic

analysis platform that uses thousands of real user inputs to instrument hundreds of apps. Third, in Chapter 6 the combined knowledge is used to prototype a solution able to instrument and improve app and device resource usage. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Background & Related Work

Before delving into the problem statement and contributions of this thesis, in this chapter, the background (Section 2.1) and related work (Section 2.2) are discussed.

2.1 Background

To better understand the coming chapters, it is important to have an understanding of Android's components and involved stakeholders. Next sections start by describing the Android architecture (section 2.1.1) and execution scheduling mechanisms (section 2.1.1). Then the Android's stakeholder ecosystem is presented, along with its fragmentation and security concerns (section 2.1.4).

2.1.1 Android architecture

The Android stack is composed by the following components:

- **Linux kernel** - On the bottom of the Android stack there is a Linux kernel which provides the system functionalities required to manage processes, memory, networks and devices. Although the newer Android versions are based on the Linux Kernel 3.4 or newer, older versions might have kernel versions starting from the 2.6. The Linux Kernel present on Android has a few Google patches on top of it that include components such as Binder (for inter-process communication), ashmem (shared memory), pmem (process memory allocator), logger, among others.
- **Hardware Abstraction Layer** - The Hardware Abstraction Layer (HAL) provides an interface between the Android system and the hardware drivers. It allows the Android system to be agnostic about the lower-level implementation of drivers and hardware.

- **Android runtime and libraries** - This layer provides useful native code libraries (e.g. OpenGL, SSL, bionic) and the register-based virtual machine - Dalvik. The Dalvik Virtual Machine (DVM) uses the Linux memory and threading mechanisms. Applications are compiled in the Dalvik Executable (DEX) format in order to be interpreted by the DVM. Each application in Android runs over a dedicated process on top of its own DVM forked from a initial process called Zygote. Each process is run in isolation generally with a different user ID (UID) and with limited file system permissions.

- **Application framework** - Both the Android framework and applications are implemented in Java and run over the DVM. The Android framework provides all the services for the applications to run. Some of these services include the Activity Manager (application/Activity lifecycle), Content Providers, Views (to create application interfaces) and the Package, Resource, Notification, Telephony and Location managers. The communication between the application framework and Android System is done through the Binder Inter-Process Communication (IPC) mechanism.

- **Applications** - An Android application generally consists of a set of loosely bound Activities. Activities are generally focused on a small set of actions the user can perform. Its user interface is composed of a hierarchy of Views that design the layout of the application. For example, a View can include interactive elements such as buttons, text input, etc. By default all the UI code runs over a single thread and all background processing is typically done by resorting to AsynsTask (for short operations that publish results to the UI), Service (long running operations in background with no UI), HandlerThread (Java thread with a message loop) or Alarms for periodically repeating actions. Android applications are generally started either due to user interaction (e.g. user click on the icon) or due to an event occurring on the OS or on another application (e.g., via broadcasts and intents) being received by a previously registered broadcast receiver (e.g., on boot event).

2.1.2 Android Execution Scheduling

Alarms are the primary mechanism Android provides to allow applications to schedule background activities. Alarms come in two flavors: 1) time critical alarms, and 2) non-time critical. The first type is called an *exact* alarm, and the second is known as an *inexact* or *deferrable* alarm. The OS is expected to execute exact alarms on schedule, but can delay the execution of deferrable alarms. Deferrable alarms are particularly interesting due to the manner in which Android can leverage them to improve power efficiency. For example,

Alarm API	SDK $<$ 19	SDK $=$ 19 - 22	SDK $=$ 23
set	Exact	Inexact	Inexact
setRepeating	Exact	Inexact	Inexact
setInexactRepeating	Inexact	Inexact	Inexact
setExact	NA	Exact	Exact
setWindow	NA	Inexact	Inexact
setAndAllowWhileIdle	NA	NA	Inexact
setExactAndAllowWhileIdle	NA	NA	Exact

Table 2.1 Behavior of alarms based on the Target SDK level. Note that although the dataset was collected before SDK 23 was available, the continuing effort put into the alarm API highlights the critical nature of Android Alarms.

batching alarms to multiplex network activity of multiple applications can reduce the wake up frequency of the device's radio.

Decisions related to what type of alarms to use are left to the application developers since, in theory, only they have the insight necessary to assess the impact a delayed alarm will have on their app. Unfortunately, developers will often optimize for profit (e.g., ensuring fresh ads are retrieved/displayed as often as possible) and usability rather than energy efficiency. A second wrinkle with alarm types is that developers are free to define what Android SDK their app targets. If the device the application is installed on has a different SDK than the targeted one, a compatibility mode applies which in some cases can alter the app's behavior. This can have interesting consequences for alarms because the default functionality for a given Alarm API call might differ between SDK versions (see Table 2.1). E.g., if the targeted SDK version is less than 19, all API calls but `setInexactRepeating` create exact alarms. For SDK 19+, a new call to explicitly create exact alarms is introduced, and the behavior for the previously existing calls is changed to create inexact alarms.

Therefore, applications with exact alarms are those which: 1) have target SDK lower then 19 *and* use `set` or `setRepeating` calls or 2) use `setExact`.

Applications with inexact alarms are those which: 1) target $\geq$ SDK 19 *and* use `set` or `setRepeating` calls or 2) use `setInexactRepeating` *or* `setWindow` calls. It is important to note, however, that despite being able to create inexact alarms for SDK $<$ 19, alarm batching across applications is only available for devices with Android KitKat (SDK 19) or higher [60].

Alternatively, Android apps can also use a new alternative designed to facilitate correct implementation of alarms and to reduce alarm occurrences based on app requirements: the

JobInfo API. The JobInfo API provides new triggering conditions based on e.g., network (metered/unmetered) and device state (e.g., idle/charging), backed by more sophisticated retry mechanisms to avoid unnecessary execution, in turn allowing tuning apps with respect to battery consumption. In this thesis the use JobInfo APIs is ommited as it was introduced 6 months prior to the experiments in Chapter 4, however, *none* of the apps in the chapter's dataset was using these APIs.

2.1.3 Android ecosystem

Android source code is open source but most Android devices in the market are sold with a combination of open and proprietary software. This software includes boot loaders, peripherical firmware, radio components, digital right management software (DRM) and applications [51]. Furthermore, its development is not fully open since its source code is often only made available after new releases. The Android firmware versions found on user mobile devices are a result of the collaboration of multiple stakeholders :

- **Google** - Android source code is released by Google under open source licenses. Google impacts the core OS on all its levels.
- **Hardware vendors** - Hardware vendors compile native binaries depending on processor architecture (e.g. ARM, Intel x86, MIPS). Different System-On-Chip (SoC) need different Linux Kernel support.
- **Original Equipment manufacturers** (OEM) - OEMs produce the heterogeneous end-user devices with varying characteristics (screen size, battery, sensors..). They introduce changes on most levels of Android, such as new kernel device drivers, proprietary bits, user-space libraries and Android framework modifications (ex: HTS Sense, Samsung touchwiz UIs).
- **Carriers** - Carriers have to comply with Google requirements in order to be allowed to include the Google Play. Carrier devices generally include carrier customized Android builds with modified boot screen, access point configurations and pre-loaded applications (installed on the system partition).
- **Developers** - Developers create new applications and adapt them to the heterogeneous devices. Applications without root permissions generally don't introduce modifications on the Android OS.

This complex ecosystem is the base to the Android fragmentation which is discuss in the following section.

Android	Codename	Distribution	Year
2.2	Froyo	0.8%	2010
2.3.3-7	Gingerbread	14.9%	2011
4.0.3-4	Ice Cream Sandwich	12.3%	2011
4.1.x		29%	2012
4.2.x	Jelly Bean	19.1%	2012
4.3		10.3%	2013
4.4	KitKat	13.6%	2013

Table 2.2 Percentage of devices running a given version of the Android Platform (June 2014).

2.1.4 Android Fragmentation

The smartphone market is largely dominated by two actors: Apple and Google. While Apple's platform is a largely controlled experience due to a strong tie between hardware and software, Google's Android is open-sourced with a soft licensing model where hardware vendors can adapt the system to their needs. This has led to Android quickly dominating the mobile market in terms of market share.

Android is the result of the collaboration between multiple stakeholders: Google, hardware vendors, OEMs, carriers, and developers. These stakeholders can customize Android with kernel patches, Android Operating System (AOS) modifications, and custom system applications. However, this collaborative ecosystem and the heterogeneous nature of Android devices makes it difficult to maintain devices and provide a consistent experience, ultimately leading to fragmentation [77, 83, 90, 106, 110, 162].

Consider Figure 2.1, which plots the observed Android version distribution of devices visiting Google Play from June 2012 until September 2014, and Table 2.2, the values relative to a 7-day period ending on June 4, 2014. Clearly, the adoption of new AOS versions is slow: up-to-date devices are in fact a minority. For example, in June 2014 there were more Android Gingerbread (14.9%) devices than Ice Cream Sandwich (12.3%) or KitKat (13.6%). Gingerbread 2.3[1] was released at the end of 2010, thus *these devices have not been updated for nearly five years*.

Part of the reason for this fragmentation is that many handsets *never receive firmware upgrades* resulting in outdated and unsupported devices after only a few months of sales. For example, the high-end AT&T and Verizon Motorola X (released in September 2014) is no longer supported after a year and will not be upgraded to Android Marshmallow [109]. Often,

[1] Table 2.2 data was collected from Google Android Dashboards over time, which only provides values for Gingerbread 2.3.3, released in February 2011.

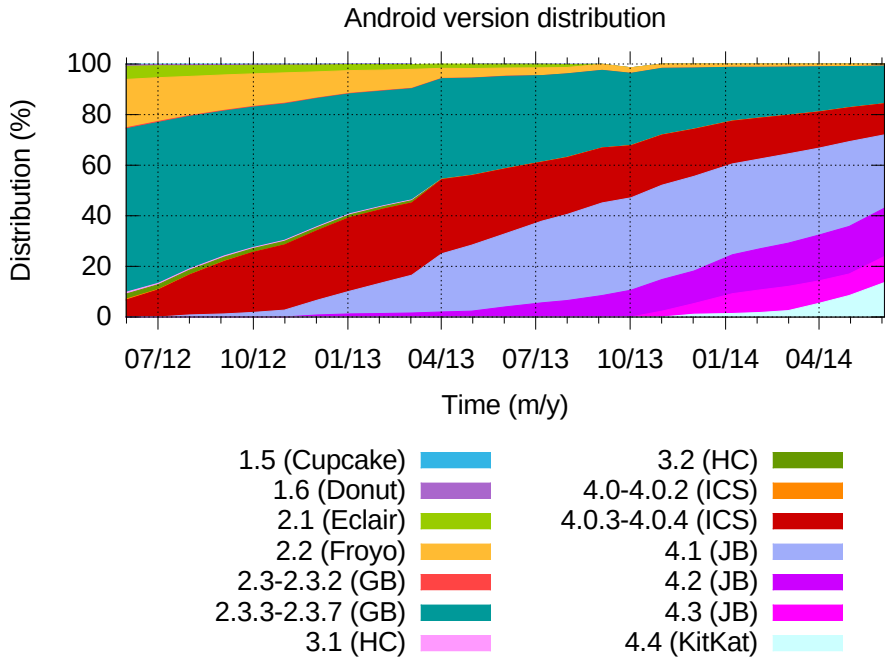


Fig. 2.1 Android version distribution from June 2012 to September 2014.

these devices are either vulnerable to security issues or suffer from critical bugs. This also affects developers since they need to target a subset of the most popular handsets to ensure adoption, while often leaving many devices unsupported. The result of this fragmentation leads to three concrete problems: 1) lack of security patches and updates, 2) hardware dependent functionality, and 3) unavailable APIs.

Security patches and updates are a major concern considering the massive Android user base[2]. Even in the presence of important security flaws, e.g., the WebView and MasterKey vulnerabilities (arbitrary code execution vulnerability affecting up to 86% of the devices), there is little incentive to maintain Android devices with the complex synchronization efforts necessary to produce updates since the current life-cycle for devices is often less than a year. In fact, Google has publicly announced that they will not back-port certain fixes to older versions of Android [107], leaving users and the community to develop fixes themselves. The lack of security updates is even more alarming considering the rise of malware on Android [105]. To accelerate the distribution and spread of security updates, Google included novel Security APIs to Google Mobile Services (GMS) in July 2014. Unfortunately, during this thesis app analysis, it was discovered that as of February 2015 only 0.48% of leaderboard applications are making use of these APIs (details provided in Section 6.3.5). To make matters

[2] 1 billion monthly users as claimed in Google I/O 2014.

worse, licensing restrictions [34, 142] prevent the inclusion of Google Mobile Services on many devices that take advantage of the rise of non-Google distributions with more frequent updates (e.g., CyanogenMod).

Hardware dependent functionality is an artifact of the competitive nature and advancements in mobile chipsets. While these SDKs can increase performance and enhance the user experience, even year old handset models can quickly be left behind. Chipset manufacturers often offer SDKs targeting specialized hardware (e.g., [125]) and can also introduce hardware specific bugs (e.g., floating-point calculation bug on MediaTek processors [64]). Vendor-specific hardware fragmentation in Android has been a topic of research [77, 90, 162], for example, [77] provides evidence of such occurrences through the use of topic analysis of Android bug reports for two Android smartphone vendors (Motorola and HTC).

Unavailable APIs are mostly a result of outdated versions of Android and Google's continuing efforts to push functionality out of the Android core and into GMS. On the surface, this does not seem like a major problem, however, OEMs must comply with a restrictive license [34, 142] to include GMS in their distributions. Among these non-compliant devices (20% of market share in 2Q 2014 according to ABI Research [7]) are those from Amazon (e.g., Kindle Fire), Nokia (e.g., Nokia X), those featuring the growing number of alternative application markets (e.g., Chinese devices such as Xiaomi, which in 2014 reached third place in global smartphone shipments after Samsung and Apple), and many Android-powered devices like car dash radio, TV sets, and photo frames. The restrictive licensing terms has even been the impetus for developing completely Google independent Android distributions [31]. Unfortunately, applications that use competing services or open-source APIs do not benefit from regular updates released via GMS.

2.2 Related Work

This section describes this thesis' related work. First, Sections 2.2.1 and 2.2.2 present the existing work on Android's execution and traffic scheduling and their energy impact. Then Sections 2.2.3 and 2.2.4 focus on the literature regarding testing automation and tester acquisition. Finally, Sections 2.2.5 to 2.2.8 introduce the literature on mobile code offloading and clone detection, dynamic software patches and industry initiatives related to offloading.

2.2.1 Android Execution Scheduling

In Android, apps and their respective alarm code can make use of `wakelocks` to prevent the system from going to sleep. Multiple papers focused on the analysis of no-sleep energy

bugs due to the use of `wakelocks` [10, 87, 88, 91, 121, 160]. Some of these used static analysis to detect occurrences, incorrect usage and placement of `wakelocks`. According to Kim et al. [91], the battery drainage due to misuse of `wakelocks` is reported to be at least 5% to 25% per hour.

Park et al. [119] proposed AlarmScope, a system to reduce non-critical alarms and reduce energy consumption. Their study of 15 Android applications, alarms are often unnecessarily set as non-deferrable. Experiments with 15 applications showed a 25% reduction of total wake-up time and energy consumption reduction of up to 12.5%. They highlight that in order to reduce energy waste, the system should not give full permission to developers. While [119] presents some interesting findings, it was limited scope and did not consider the dependency of the alarm API behavior on the application target SDK.

2.2.2 Android Energy Consumption

Mobile traffic is typically scheduled using static timers (Radio Resource Control inactivity timers), which often leads to the tail times of resource allocation even after all data was transmitted. Due to the expensive characteristics of mobile data, optimizing the use of radio resource allocation windows has been a hot topic in the past. Mobile data energy consumption from the perspective of end user equipment has been addressed via traffic shaping techniques [9, 26, 33, 47, 81, 82, 98, 123, 123, 135, 150] (e.g., traffic batching and tail-time reduction), these approaches have been shown to have severe limitations. Most of these focused on: coalescing data transfers [33]; pre-fetching [124]; batching and tail time sharing [47, 123]; and fast-dormancy [26, 81, 82, 123].

However, Vergara et al. [150] recognized that most traffic shaping algorithms (aggregation and delay) proposed in literature [26, 33, 79, 98, 112, 149] were based on traffic simulation using pre-recorded packet traces. It claims that these approaches ignore application-protocol interactions and do not measure the energy consumption of the respective solutions. They concluded that performing traffic shaping independently of application behavior can be counterproductive (e.g., Skype traffic increased by 22x).

Thus applying these techniques without application and operating system integration can often *increase* energy consumption due to retransmissions and/or signaling [150], and many works [29, 123, 135, 150] have highlighted the need for better application knowledge and/or integration with OS/platforms. Because of this limitation, the proposed delay mechanisms are often restricted to relatively short periods of time. Furthermore, these solutions often require applications to set an interval of delay tolerance [135] when attempting to send information, which means that to support such behavior, apps need to be modified and based on the results in [119], there is no reason to believe developers would properly configure things anyways.

2.2.3 Automated App Testing

As mobile apps have grown in ubiquity and importance in our daily lives, there has been increasing focus on them by the research community. In Chapter 5, CHIMP is presented, a tool to test mobile apps and gather real user inputs at scale. While CHIMP draws heavily from the automated testing literature, it is designed to meet the needs of researchers in a wide variety of contexts. This section provides background on the state of the art in automated testing tools as well as an overview of other app behavior measurement techniques and applications.

Automated testing can be considered a search problem where the objective is to "explore" the largest possible set of app functionalities within a defined time span. Such an exploration is usually measured in terms of *code coverage*, i.e., the number of lines of code in the target app that the test exercises. In [40] authors review the state of the art, studying and evaluating 14 testing tools grouped into 3 categories: *random*, *model based*, and *systematic*.
Random tools [61, 100, 132, 158] are best exemplified by the official Android monkey [61]. They amount to a blind search through the app being tested. Random testing tools are reasonably easy to use and often provide pretty good coverage.

Model based [16, 32, 39, 78, 156] tools view mobile apps as a finite automata where user actions trigger transitions between states. Models can be extracted considering the sequence of function calls (call graph model - CGM), the user interface layout, and interaction between components (interface model - IM). After the model is built, the testing corresponds to exploring (with varying degrees of sophistication) the space of the state machine and terminating when all possible state transitions have been discovered.

Systematic exploration tools [18, 32, 101, 147] are more complicated and use things like evolutionary algorithms in an attempt to produce inputs that cover an app. EvoDroid [101] is an example. It uses the IM as "genes" and the CGM as space to be explored while a fitness function tries to optimize code coverage and guide the exploration. The most novel aspect of EvoDroid is that, unlike previous genetic algorithm approaches for testing, the CGM is broken up into *segments*, each of which are tested separately. Thus, the fitness function for a set of tests running on a segment will heavily favor the tests that are able to reach the next segment. When a new segment is chosen for testing, the genes from the previous segment are prepended, eventually resulting in tests that achieve maximum code coverage.

All these tools have strengths and weaknesses, but ultimately [40] finds no tool to be superior; indeed, monkeys often beat more sophisticated tools in terms of code coverage. They share however an important limitation: they are "stress tests" tools only. Since no real human input is synthesized, no information regarding actual human behavior is collected (i.e., how users react to a user interface), nor if users consider the app performing correctly.

There are also tasks that might either be easier for, or even *require*, real humans, e.g., login screens, forms, games, etc.

There are several services out there that can bring humans into the loop of app testing, similarly as CHIMP does. The two most popular services are offered by Amazon [17] and Google [66], and integrated into their app stores. CHIMP is different in a few ways though. First, they are meant to be used by an app's developer exclusively, and therefore not good candidates for large scale app analysis. Second, they provide two mechanisms for selecting testers, either developer provided mailing lists or making a version of the app available for open beta testing in their stores. Finally, these tools are mostly oriented towards testing app compatibility with different devices and do not provide any additional access or low level information (e.g., traffic dumps, instrumentation, etc.) which researchers struggle to capture at scale. Like CHIMP, Appetize [24] provides streaming of mobile devices to browsers. However its focus is not measurements and experimentation, for example, it does not provide any mechanism to acquire users or to analyze apps at scale.

2.2.4 Crowdsourced Systems

Although not quite an automated testing tool, Varvello et al. [148] built EYEORG, a platform for crowdsourcing web quality of experience measurements. EYEORG presents paid crowdsourced workers with interactive videos of web page crawls, allowing users to provide judgments on performance in a controlled, yet scalable environment. CHIMP is similar to EYEORG in spirit but it provides a completely orthogonal service. Nevertheless, CHIMP's evaluation follows the validation methodology laid out by [148], which includes using engagement estimation, and control questions (§5.4).

Nikravesh et al. built Mobilizer [113], a platform for performing network measurements in a mobile environment that also leverages crowdsourcing. The key insight of Mobilyzer is that the idea of a "killer app" that can reach enough user penetration to be of meaningful use is not very realistic. Mobilyzer is delivered as a combination of library and service. Experimenters can design and issue network experiments to gain a view of network conditions across all Mobilyzer devices. They demonstrate ease of development ("about an hour" for a 3rd party throughput-testing app's developers to integrate Mobilyzer) and demonstrate its effectiveness as a measurement platform by conducting crowdsourced measurements of mobile Web performance and Video QoE.

While obviously related, Mobilyzer and CHIMP set out with fundamentally different objectives. While Mobilyzer does a great job at providing a previously unseen global view of the mobile network, CHIMP focuses more on app and user behavior, which gives researchers and developers a different view of the mobile environment.

MCO	Partitions	Dynamic	No Repackage	Stock OS	Off–Cloud	Off–Network	Deployment
MAUI [45]	Manual / Method	✗	✗	✓	✓	✗	✗
ThinkAir [93]	Manual / Method	✗	✗	✓	✓	✗	✓(EC2,cost)
CloneCloud [41]	Auto / Thread	✓	✗	✗	✓	✗	✗
Comet [73]	Auto / Thread	–	✓	✗	✓	✗	✗
Zhang et al. [161]	Auto / Class	✗	✗	✓	✓	✗	✗
INFv	Auto / Class	✓	✓	✓	✓	✓	✓(cache,load)

Table 2.3 Mobile Cloud Offloading (MCO) systems and properties. The comparison reveals that INFv supersedes the previous designs in many aspects. (red cross means the feature is not supported whereas green tick means the opposite.)

CHIMP is designed to complement the state of the art tools, and to offer a flexible platform to collect a rich set of output data targeting specifically human behavior. For app developers, it means that they can quickly A-B test design and algorithmic choices before releasing an app to an app-store. For the research community, it means it is now possible to run experiments on apps the researcher has no control over. Indeed, the impetus for building CHIMP was our frustration as researchers when trying to collect mobile app interaction data for even dozens of users, let alone hundreds or thousands.

2.2.5 Mobile Code Offloading

In Chapter 6, INFv is proposed, a non-intrusive in-network Mobile Code Offloading (MCO) solution. MCO is a reasonably well explored area (see Table 2.3), however earlier work has a few limitations that this thesis directly addresses with INFv. This section provides an overview of previous work, focusing on the lessons taken away that were used to design INFv. MCO systems can be differentiated based on their granularity and partitioning decisions (what to offload), offloading techniques (how to offload), and runtime decisions (when to offload).

What to offload can be defined in a *manual* (app developer assisted) or in an *automated* manner. The first can be accomplished via programming frameworks and/or code annotations. For programming frameworks [38, 89, 93], both local and remote execution alternatives have to be implemented according to the framework's design constraints (e.g., concurrency models). In Maui [45], annotations allow a partially automated offloading solution where developers select methods to offload. The benefit of these explicit systems is the level of customization; developers have a large degree of control over how their apps are offloaded.

An alternative approach taken by other work is to make automated offloading decisions [41, 73, 161] by performing static and dynamic analysis of apps. While automated approaches give up a degree of flexibility, they benefit from being able to leverage the existing

app ecosystem and general ease of use. That said, these systems do not really focus on the deployment characteristics of offloading. Thinkair [93] and Cloudlets [133], however do to some extent. Thinkair allows on-demand execution in a cloud environment. It provides 6 different VM types, with varying CPU and memory configurations. Mobile devices upload specially crafted apps to the cloud, and their local counterpart negotiates the on-demand execution on one of these VMs. A more robust approach (one that INFv has taken) is to support existing apps while handling resource negotiation and functionality caching in a fully automated and transparent manner. In particular, INFv tries to support heterogeneous network topologies and load balance cached functionality in an intelligent manner. Cloudlets [133] in particular serves as a motivation for INFv as it highlights the impact of high latency in MCO to justify the need for deploying physically proximate decentralized clusters to execute functionality. INFv directly addresses the problems and challenges raised in this work by proposing an in-network solution ("Network" in Table 2.3) for MCO.

The majority of offloading literature proposes a method offloading granularity [38, 45, 89, 93]. CloneCloud [41] and Comet [73] propose full or partial thread granularity. Automated method granularity architectures incur the cost of synchronizing the serialized method caller objects, parameters, changed state and return object. Thread granularity architectures often need to synchronize thread state, virtual state, program counters, registers, stack or locks. INFv is the first to address the challenges behind caching app functionality, complementing its granularity design with a real market study to reduce its app storage requirements.

How to offload depends on the aforementioned characteristics. Manual approaches tend to use custom compilers (e.g., AspectJ) or builders. Automated solutions often operate on compiled apps and rely on byte-code rewriting [161] or VM modifications [41, 73]. Altering an app generally requires repackaging and resigning it, which also implies the need for a new distribution mechanism incompatible with current app markets. Each of the above architectures, except for Comet [73] (a distributed shared memory solution), need either app repackaging, re-writes, or both, impacting their likelihood of adoption. INFv offloading differs mainly in that it does not require a custom distribution, manual intervention or app-repackaging even in the presence of app updates. It allows for dynamically loaded partitions ("Dynamic" in Table 2.3) and is fully reversible.

When to offload was mostly done using thresholds [73] or Integer Linear Programming [41, 45] based on the app profiling metrics.Similar to CloneCloud, INFv relies on UI instrumentation to profile the different execution paths of apps. INFv performs app profiling on remote servers to reduce the overhead on mobile devices and like Thinkair [93], its energy model is based on PowerTutor [160]. INFv improves previous systems by offloading together functionality with high communication based on their runtime invocation frequency.

2.2.6 Mobile Edge Computing

Mobile Edge Computing [56] (MEC) is an industry initiative[3] whose goal is to provide computing capabilities at the edge of the cellular network. Its focus is explicitly on the infrastructure and deployment and not on the potential applications. That said, INFv can be considered an obvious use case of MEC and the first full-fledged MCO architecture to exploit its potential.

2.2.7 Runtime Patching

Runtime patching has been used to dynamically provide updates to apps. OPUS [15] focused on providing dynamic software patching to C programs, while POLUS [37] was more focused on updating long-lived server side apps. More recently, such techniques were brought to Android with PatchDroid [110]. It focused on security vulnerabilities and proposed a system to distribute and apply third-party in-memory security patches. Inspired by these systems, INFv modifies a single Android OS binary to extends app's functionality at runtime, providing mechanisms similar to those of aspect oriented programming.

2.2.8 Clone detection

Clone detection literature [48, 97, 108, 129] focused on detecting app cloning using class/method names and tend to ignore minor custom changes to the functionality. In code caching, one is more interested in the unmodified use of third-party libraries than similar code. Unlike recent studies [97, 108, 152], some initial works [48, 129] did not consider obfuscation, which can impact the statistical significance of their results. In [108] and [97] the authors study obfuscation based on class names. Unfortunately, they do not consider package obfuscation which affects the offloading routing mechanisms. Wukong et al. [152] focused on clone detection based on Android API calls, and while it highlights the challenges in overcoming obfuscation, this thesis shows strong evidence that, a simple package name filtering might suffice to detect obfuscation at a package level.

[3] supported by Huawei, IBM, Intel, Nokia, NTT DOCOMO and Vodafone.

Chapter 3

Problem Statement & Contributions

The focus of this thesis is on the study and development of techniques to detect and mitigate computation and energy inefficiencies in mobile apps. Hereby the main research questions and contributions to achieve this goal are presented.

First, Section 3.1 describes the challenges in studying **what** apps can do from the perspective of execution scheduling. It studies apps' structure and impact of developer decisions and OS fragmentation in the app resource usage efficiency. Second, Section 3.2 delves into the challenges behind analyzing **how** apps are executing and how to test them with realistic human-like inputs at scale. Finally, using the combined knowledge, Section 3.3 focuses on the challenge of detecting **when** these executions can be optimized in terms of energy consumption. These three sections represent a chapter of the thesis each, i.e., Chapters 4, 5 and 6, respectively.

3.1 Android Execution Scheduling

Todays mobile devices support a diverse set of functionality, much of which is not dependent on active user interaction. Many tasks are performed in the background, which has very clear impact on battery life and mobile data usage [29]. *Alarms* are Android's integrated application execution scheduling mechanism (used, e.g., for background network activity) and are a primary vehicle for improving devices energy efficiency. One way Android mitigates Alarms' negative impact is *batching*, which can reduce total device awake time while increasing the chance that traffic from different applications can occur simultaneously (and thus greatly reducing energy consumption [26, 33, 79, 98, 112, 149]). Unfortunately, the success of batching depends on the correct usage of alarm APIs (by applications) which widespread and impact are unknown.

Therefore the first research question is:

Q1: How widespread is alarm usage across apps and what is its impact on execution efficiency?

Upon studying the usage of alarms across apps it is important to (**Q1.1**) detect whether they are being misused and how. And finally, if its (**Q1.2**) misuse is due to inconsistencies in the Android OS or due to bad development practices. In the case of Android OS inconsistencies, how does it relate to the OS fragmentation problems described in Section 2.1.4.

3.1.1 Methodology

To answer these research questions, over 22 thousand of the most popular apps of the Google Play market were crawled, downloaded, decompiled and statically analyzed. The analysis consisted of building and traversing the app static call graph (in this case, vertices and edges represent, respectively, methods and invocations), detecting the usage of alarm API calls along with common Android build parameters. Finally, two sets of 30 apps were selected – popular and alarm misusing – and executed to demonstrate how static analysis can be used to detect alarm API misuse and measure the correlation between the number of defined alarms and device's wakeup frequency.

3.1.2 Contributions

- Through a measurement study, it is shown that many of the app execution scheduling (alarms) misuses can be detected by crawling existing markets, downloading and statically analyzing apps' code (i.e., detecting API usages and build properties).
- A new facet of the fragmentation problem is revealed and demonstrated: even if the device is supported and up-to-date, apps often target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device.
- It is shown that alarms are largely present in Android apps and are often misused, greatly impacting devices wake up frequency and the efficiency of the existing energy saving techniques (e.g., batching). It indicates that research on energy efficiency on mobile devices needs to incorporate an understanding around the use of alarms.

3.2 Large Scale Dynamic Analysis

While static analysis is widely used in security [104] and computation offloading research [161], it studies **what** an application can do but fails to capture app's runtime behavior. Dynamic

properties such as the app's dynamic execution control flow (**how** the app executes) and invocation counts are crucial for understanding the resource requirements of apps. Unfortunately, realistic app testing is still challenging at scale with existing solutions often relying on distributing specially instrumented apps, or even phones with pre-installed apps, to real users. This approach is quite expensive and has intrinsic scalability limitations, so the second research question is:

Q2: How can one gather realistic user inputs for mobile device apps at scale?

Given that such a platform can be built to include real users in the app testing loop (without requiring installing apps or giving away phones), it requires mechanisms to evaluate (**Q2.1**) how users perform and how do they differ from existing UI automation tools.

Existing code automation literature [16, 18, 39, 40, 100] relies on runtime tracing (i.e., logging method calls) to benchmark their tools in terms of code coverage. Unfortunately these studies are generally small scale and their tracing tools rely on having access to the apps source code, i.e., they are restricted to unrealistic apps with orders of magnitude less complexity than Android's popular apps. This thesis aim is not only to provide realistic app inputs but also to explore (**Q2.2**) how to analyze realistic closed-source apps behavior at runtime (including code tracing, coverage and similarity) at scale.

Finally, given the hypothesis that the previous challenges can be overcome, two questions arise: (**Q2.3**) what would be the differences and combined gains of using existing UI automation tools and humans (e.g., do they achieve higher code coverage?); and if the combined knowledge could be used to build better UI automation tools.

3.2.1 Methodology

First, an architecture for streaming virtualized Android apps to browsers was designed so users can test apps without requiring a mobile device. Then this platform was integrated with existing crowdsourcing platforms (crowdflower) to provide the required users for testing apps. To enable tracing of realistic apps, a state of the art module was proposed to interact with the Android's app debugger port and retrieve runtime method traces (i.e., method invocation call graph as well as information regarding including and exclusive execution times) even for closed-source apps.

Traces were parsed to determine the executed method calls, which are then compared to the method calls from the app retrieved via static analysis (i.e., by parsing the decompiled app code), outputting the percentage of code covered. Additionally Jaccard indexes were used to compare different trace method calls and estimate their similarity.

To answer the aforementioned research questions, thousands of apps were tested with the defacto standard UI automation tool ("monkey") and thousands of real users (humans).

Initial tests were performed to design guidelines for analyzing and validating user behavior. Then a detailed behavior comparison was performed between humans and the "monkey" UI automation tool, in terms of code coverage, similarity and generated network traffic (one of the major culprits of energy consumption in mobile devices).

Finally, to demonstrate how human input, priorly difficult to collect at scale, can lead to better research tools, two modules were built on top of the platform: a traffic classification and a recurrent neural network (RNN) based UI automation modules. The first was tested based on its f1 score, while the second based on the number of clicks and time required to achieve a common task by the "monkey" tool versus the RNN prototype.

3.2.2 Contributions

- A solution for enabling large-scale closed-source app testing and runtime tracing based on a combination of crowdsourcing and mobile app streaming was presented and evaluated, along with the key design principles that allow researchers to collect human inputs, priorly difficult to collect at scale.
- Via a measurement study it is shown that introducing humans in the app testing loop improves the coverage of apps' code and generated network volume. Unlike previous UI automation literature which relied on simple and unrealistic open-source apps, this study uses VM profiling to analyze the execution of popular closed-source apps.
- It is shown that, while the defacto UI automation tools are unable to generate sufficient traffic volume, human generated traffic can be used to build network traffic classifiers (random forest) to classify the traffic according to its originating app, reaching high f1 scores even in the presence of encrypted traffic.
- Finally, a promising future direction is proposed, where a RNN-based prototype trained with human interactions is used to improve over the existing UI automation tools, reducing the number of clicks and time required to achieve common tasks.

3.3 Dynamic Code Execution and Offloading

Using the combined knowledge of Sections 3.1 and 3.2, this section focuses on the challenge of detecting **when** these executions can be optimized in terms of energy consumption.

Recent work has proposed various solutions to detect, offload and execute computationally intensive functionality of mobile apps remotely in a cloud, referred to as a mobile-to-cloud paradigm [38, 41, 45, 73, 89, 93, 161]. Their evaluations have shown that the energy

consumption of CPU intensive apps, e.g., multimedia processing apps and video games, can be reduced by an order of magnitude [41, 45]. Beside the extended battery life, there are other benefits, such as faster execution time, responsiveness, and enhanced security by dynamic patching [110]. Unfortunately, these works either failed to address the challenges of deploying and scaling mobile code offloading systems at all, or overlooked the opportunities to effectively exploit in-network resources. Furthermore, these solutions utilize intrusive offloading techniques which either require custom OS distributions, app repackaging, or even alternative app stores, which not only increases security risks and deployment costs, but also greatly increases the barrier to market adoption. Finally, while they focus on selecting a partition of app's code to run in the cloud, they did not measure the applicability of their partitioning algorithms to real apps nor the cost of hosting the most popular apps.

So the third research question arises:

Q3: How can offloading occur without requiring intrusive techniques (e.g., OS distribution, app repackaging, alternative stores, etc) and can it still provide the offloading computational and energy benefits?

Considering a potential solution and the lack of real-world validation, other important questions arise: (**Q3.1**) how applicable are its partitioning mechanisms to the most popular apps; and (**Q3.2**) what is the storage cost of hosting the most popular functionality. Additionally, it would be interesting to find if such storage can be reduced.

While there is research [48, 97, 108, 129] on detecting app cloning using class/method names, when considering code caching, one is more interested in the unmodified use of third-party libraries than similar code. Unlike recent studies [97, 108, 152], some initial works [48, 129] did not consider obfuscation, which can impact the statistical significance of their results. So, as a result of **Q3.2**, estimating the possible storage reduction not only requires a different approach from the clone detection literature but it is also important to (**Q3.3**) estimate the impact of obfuscation in the analysis of code reuse.

3.3.1 Methodology

A new offloading system was built that integrates with the previous static and dynamic analysis solutions. To avoid modifying app binaries and change apps' signatures, it implements a new offloading mechanism that targets the minimal set of changes required to provide dynamic instrumentation – a reversible binary patch to Android's *app_process* (details in Section 6.2.1). Additionally a computation distribution module was implemented that allows it to efficiently load balance computation in the network.

To evaluate the system, an experiment setup using real mobile devices as clients was used, where apps were partitioned and partly executed on a remote server. The remote counterpart

executed the partitions in an Android VM and multiple connectivity setups (wifi, 3G and 4G) were tested.

To answer **Q3**, a benchmark of the system's offloading capabilities was done using two common computation intensive apps used in previous literature. The energy consumption (using a monsoon power monitor) and execution time of both the apps were measured, with and without offloading, and using different partition strategies to depict their impact on the offloading performance. Then the cost of state synchronization between the local device and remote Android VM was analyzed. To do so an app that detects faces on images was used. The images processed varied in size to depict the trade-offs between the offloading computation speedup and the required communication cost. The system runtime decisions were also evaluated with varying network conditions (by inducing latency in the last hop before the remote server) to see if the system is able to dynamically decide whether or not it should offload the functionality. Its offloading decisions are compared against the baseline energy and execution times of executing locally.

From the network perspective, its computation load balancing mechanisms were tested through simulation (using Icarus [130]) over an ISP topology (Exodus) by varying the computation load while observing its responsiveness to the workload jitter.

To answer **Q3.1**, 24 of the top Android apps were dynamically analyzed (using the previously described runtime tracing mechanisms) regarding their possible offloading partitions (i.e., reducing the communication cost and avoiding offloading code that requires local resources). To this end two community detection algorithms were used – Girvan-Newman [59] and Louvain [35] – over the runtime call graph (where vertices and edges are, respectively, classes and method invocations) with the runtime method invocations as weights. The first algorithm returns as many partitions as specified via parameter, while the second is able to estimate the optimal number of partitions based on modularity optimization. The resulting partitions are analyzed in terms of how many communities can be offloaded (i.e., do not contain classes that use local resources, e.g., sensors) and used to provide an estimate of what percentage of the popular apps code can actually be offloaded while reducing the communication cost.

Finally, in response to **Q3.2** and **Q3.3**, over 20K apps were statically analyzed in regards to their size, percentage of unique classes and packages, and obfuscation properties.

3.3.2 Contributions

- A solution for in-network offloading of mobile app computation is presented and evaluated. It targets a single Android binary (e.g., via a fully reversible binary patch) from which all apps' processes are forked, extending them to provide offloading and

improving their execution time and energy consumption without relying on a custom OS or app modifications.

- Unlike previous solutions, this work has a special focus on in-network computation offloading and distribution, it proposes and compares different strategies to effectively balance functionality load while reducing both end-user-experienced latency and request drop rates.
- Via a large measurement study of over 20K of the most popular Google Play apps, it is shown that there is an high amount of code reuse and that there is potential to greatly reduce the costs of hosting the most popular apps.
- Via a runtime tracing measurement study of 24 of the top market apps, it is shown that community algorithms can be used to partition distinct app functionality and the amount of code eligible for offloading (i.e., with no access to local hardware resources) is quantified.

3.4 Publications

The diagram in Figure 3.1 shows how the distinct chapters of this thesis relate to the aforementioned research questions and to publications. Note that the blue circles represent the publications included in this thesis (Section 3.4.1), while purple ones represent publications not included in the thesis (Section 3.4.2). Each research question is addressed in a separate chapter, i.e., research questions **Q1**, **Q2**, and **Q3**, are addressed, respectively, in Chapter 4, 5 and 6.

3.4.1 Thesis Publications

The following publications are included in this thesis:

1. **An Empirical Study of Android Alarm Usage for Application Scheduling**.
   Almeida, Mario and Bilal, Muhammad and Blackburn, Jeremy and Papagiannaki, Konstantina. In proceedings of the 17th International Conference of Passive and Active Measurements, 373–384 (PAM '16).
   DOI: 10.1007/978-3-319-30505-9 [12]. Core Ranking B.

2. **CHIMP: Crowdsourcing Human Inputs for Mobile Phones**.

Almeida, Mario and Bilal, Muhammad and Finamore, Alessandro and Varvelo, Matteo and Grunenberger, Yan and Blackburn, Jeremy. In Proceedings of the 2018 World Wide Web Conference, 45–54 (WWW '18).
DOI: 10.1145/3178876.3186035 [13]. Core Ranking A*.

3. **Diffusing Your Mobile Apps: Extending In-Network Function Virtualisation to Mobile Function Offloading**
   Almeida, Mario and Wang, Liang and Blackburn, Jeremy and Papagiannaki, Konstantina and Crowcroft, Jon, *Diffusing Your Mobile Apps: Extending In-Network Function Virtualisation to Mobile Function Offloading*, under submission.

3.4.2 Other Publications

Other publications in which the author was involved during this thesis (ordered by date):

1. **Dissecting DNS Stakeholders in Mobile Networks**.
   Almeida, Mario and Finamore, Alessandro and Perino, Diego and Vallina-Rodriguez, Narseo and Varvello, Matteo. In proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies, 28–34 (CoNEXT '17).
   DOI: 10.1145/3143361.3143375 [14]. Core Ranking A.

2. **C3PO: Computation Congestion Control PrOactive**.
   Wang, Liang and **Almeida, Mario** and Blackburn, Jeremy and Crowcroft, Jon. In Proceedings of the 3rd ACM Conference on Information-Centric Networking, 231–236 (ACM-ICN '16).
   DOI: 10.1145/2984356.2988518 [153]

3. **Stweeler: A Framework for Twitter Bot Analysis**.
   Gilani, Zafar and **Almeida, Mario** and Farahbakhsh, Reza and Wang, Liang and Crowcroft, Jon. In Proceedings of the 25th International Conference Companion on World Wide Web, 37-38 (WWW '16 Companion).
   DOI: 10.1145/2872518.2889360 [58]

4. **RILAnalyzer: a comprehensive 3G monitor on your phone**.
   Vallina-Rodriguez, Narseo and Aucinas, Andrius and **Almeida, Mario** and Grunenberger, Yan and Papagiannaki, Konstantina and Crowcroft, Jon. In Proceedings of the 2013 conference on Internet measurement conference, 257-264 (IMC '13).
   DOI: 10.1145/2504730.2504764 [146]. Core Ranking A.

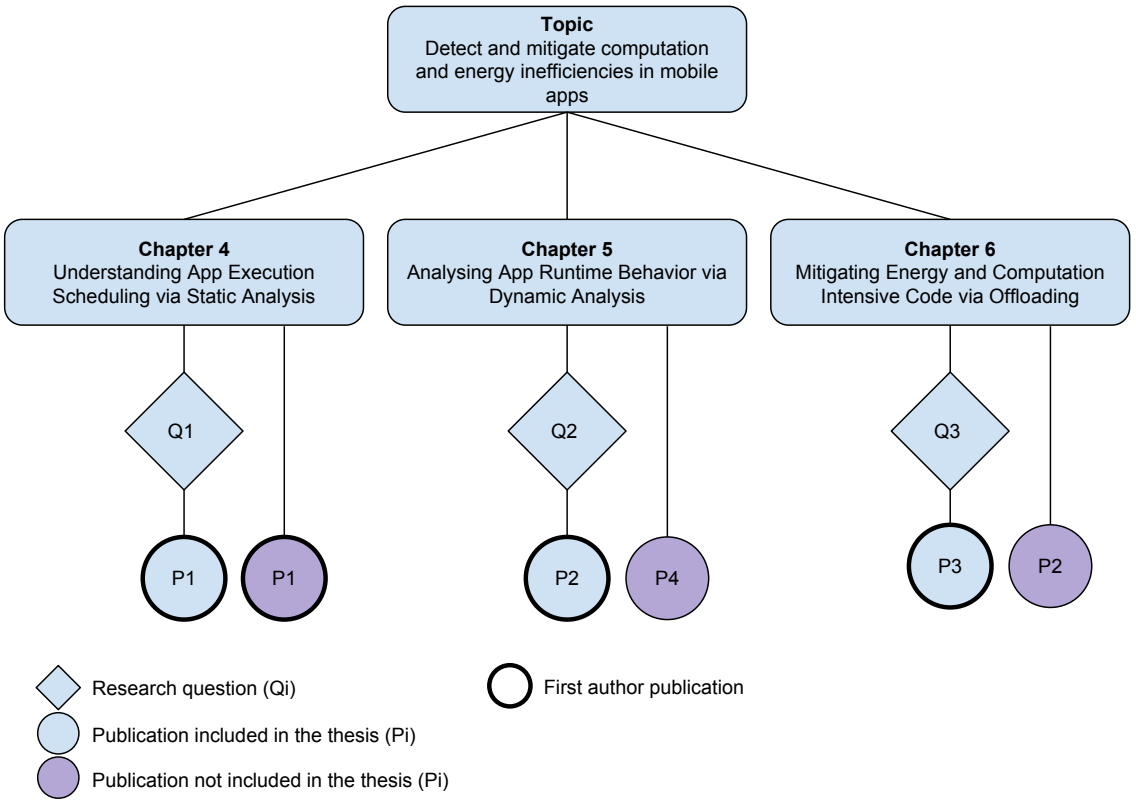


Fig. 3.1 Overview of the distinct thesis chapters and related research questions and publications.

In these publications, the following contributions were made:

- In P1, a similar approach to Chapter 4 was used to detect network/communication API usages. Additionally, it performs a measurement study of how several DNS dynamics can reduce user experience in mobile networks and shows that the ephemeral behavior of network flows results in wasted resources due to devices limited caching potential.

- In P2, an in-depth description and evaluation of the network computation and distribution mechanisms described in Chapter 6 is provided.

- P3 is unrelated to the topic of this thesis and proposes a solution for detecting and characterizing bots in social networks.

- While Chapter 6 focus mostly on computation, P4 looks at the energy impact of device's network properties. A solution is provided to allow researchers to collect low-level radio information and accurate cellular network control-plane data, as well

as user-plane data. Through a measurement study it infers how different network configurations interact with app logic, often causing network and energy overheads.

Chapter 4

Analysis of Android Execution Scheduling

Android applications often rely on alarms to schedule background tasks. Since Android KitKat, applications can opt-in for *deferrable* alarms, which allows the OS to perform alarm batching to reduce device awake time and increase the chance of network traffic being generated simultaneously by different applications. This mechanism can result in significant battery savings if appropriately adopted.

In this chapter a large scale study of the 22,695 most popular applications in the Google Play Market is performed to quantify whether expectations of more energy efficient background app execution are indeed warranted. A significant chasm between the way application developers build their apps and Android's attempt to address energy inefficiencies of background app execution is found. Along with the finding that close to half of the applications using alarms do not benefit from alarm batching capabilities. The reasons behind this is that (i) they tend to target Android SDKs lagging behind by more than 18 months, and (ii) they tend to feature third party libraries that are using non-deferrable alarms.

4.1 Introduction

Todays mobile devices support a diverse set of functionality, much of which is not dependent on active user interaction. Many tasks are performed in the background, which has very clear impact on battery life and mobile data usage [29]. The impact is substantial enough that reducing and mitigating it has been the focus of a significant amount of research and development.

A promising set of solutions aim to shape applications' traffic [26, 33, 79, 98, 112, 149], but suffer from severe limitations. These techniques ignore application-protocol interactions and lack integration with applications and OSes, often *increasing* energy consumption due to retransmissions and/or signaling issues [150] in real-world scenarios. Other works [29, 123, 135, 150] highlight the need for better application knowledge and/or integration with OS/platforms.

Alarms are Android's integrated application execution scheduling mechanism (used, e.g., for background network activity) and are a primary vehicle for executing the traffic shaping techniques. Alarms are so critical to the functionality of Android that they have been a hot topic at the last two Google IO conferences and a popular target for energy concerns[1][2]. One way Android mitigates Alarms' negative impact is *batching*, which can reduce total device awake time while increasing the chance that traffic from different applications can occur simultaneously. As of KitKat, developers can opt-in to have their alarms be *deferrable* which makes batching by the OS easier.

Unfortunately, the success of batching depends on the correct usage of alarm APIs by applications: apps themselves determine the deferrability, trigger time, and repetition interval of alarms. This leads to the situation Park et al. [119] discovered in their study of 15 Android applications: alarms are often unnecessarily set as non-deferrable. However, it is totally unclear how widespread such a practice is and thus its impact on the efficacy of alarm scheduling is unknown.

Since there is no indication that alarms will cease to be the preferred application level scheduling mechanism within Android, future design and development should be informed with an understanding of how developers use the current alarm system. Thus, in this chapter a large-scale study of 22,695 real applications from the Google Play Market (to the best of our knowledge, the largest such study to date) is performed in order to find evidence of alarm API adoption delays and their impact on the performance of the Android OS; more specifically, the effectiveness of alarm batching in Android. It investigates how many apps use alarms, what type of alarms they use, differences in alarm usage by application category, and whether alarms are being used by apps themselves or by 3rd party libraries. It is found that a shocking 46% of apps with alarms do not take advantage of Android alarm scheduling capabilities due to either targeting old SDK versions or their use of 3rd party libraries. It further discuss and analyze the problems behind Android SDK adoption and propose possible directions for improving alarm batching across applications.

[1] GIO'15, Doze - http://goo.gl/KEJURc
[2] GIO'14, Project Volta - https://goo.gl/aebnwF

4.2 Alarm Study

4.2.1 Dataset

To understand the use of alarms in Android apps, Google Play up to 564 of the most popular free apps for each Google Play category were crawled and downloaded. Removing duplicates the dataset is left with 22,695 unique apps. Although studying the most popular apps is clearly biased, it is justified for two reasons. First, these apps are more likely to be optimized than the least popular apps due to their associated revenues. Second, since these apps account for the majority of downloads, they are more representative of what users actually have on their mobile devices. This is evidenced by Viennot et al. [151] who found that the top 1% of most downloaded apps account for over 81% of the total downloads in November 30, 2013. To the best of our knowledge, the dataset (May 2015) should account for around 1.5% of the total apps of the market in 2015 (AppBrain[3] claims around 1.5 Million apps in the first quarter of 2015).

For each of the 22,695 apps, the manifest is extracted; an XML file that contains application meta-data, such as the application package name, components, permissions, etc. Three of the properties listed in the manifest are the minimum, maximum, and target SDK. The target SDK is the Android API level (e.g., Android 4.4 Kitkat has an API level of 19) that the application was developed for, and, as discussed earlier, determines the types of alarms available to the developer. By default, apps that do not define a target SDK have their target default to the minimum SDK.

4.2.2 Static Analysis

Since the focus of this study is understanding how alarms are being used by apps, static analysis is performed on the crawled apps. First they are decompiled into assembly-like code (smali). Then the smali code is statically analyzed to locate occurrences of Android Alarm API calls. In the database, each occurrence of an alarm/jobinfo API call is registered along with the respective application, alarm API, smali file name, line and annotations to the method where it occurs. Since some free apps are likely to have ads [76], opposed to their paid version, the API call locations are used to correlate them with the ad libraries (Section 4.2.5). Annotations are useful since specific methods can use the `TargetAPI` annotation to denote that they want to execute the method in compatibility mode. For apps, the meta-data (target SDK, internet usage, category) is registered. In particular, one is interested in correlating

[3] http://www.appbrain.com/stats/number-of-android-apps

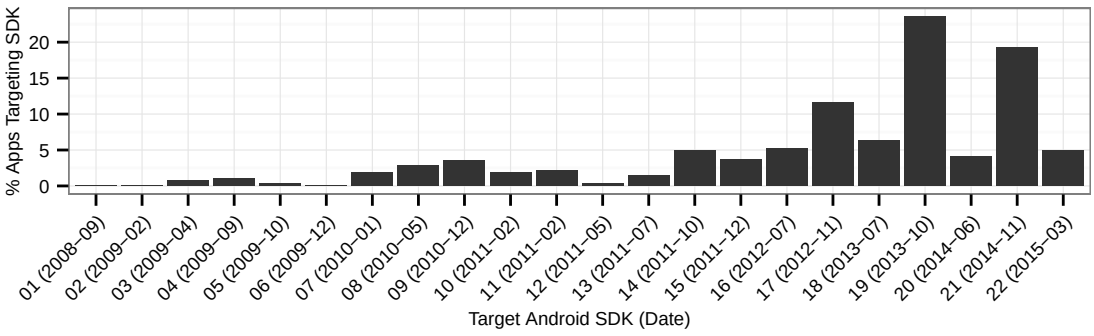


Fig. 4.1 Percentage of apps that define each Android SDK as the target SDK in their manifests. NB: The Android target SDK extraction fails for 1.5% of apps in the dataset.

Alarm Type	SDK $<$ 19	SDK $>=$ 19
AlarmInexact	8.49%	52.91%
AlarmExact	44.05%	2.31%
Alarm	46.06%	53.49%

Table 4.1 Fraction of apps with exact and inexact alarms grouped by SDK version. Dates represent the release dates of each Android SDK. Note that an application can make use of both exact and inexact alarms.

target Android SDKs with the the number of alarm API calls and their usage within different apps and app categories.

4.2.3 Impact of Target SDK on Alarms

As mentioned previously, the target SDK of an application can significantly affect the behavior of its alarms. As a first step towards understanding the impact of the chosen target SDK, Fig. 4.1 plots the distribution of SDK targets from the dataset. It shows that despite the efforts of Google to promote the use of their newer SDKs (e.g., Google IO conferences, extensive documentation and application design guidelines), the majority of the popular apps target SDKs that were released more than 18 months ago (up to and including SDK 19, represent 71.6%). Close to half (48%) the apps target SDKs lagging behind by more than 21 months.

From the perspective of alarms, note that 47.23% of apps have a target SDK lower than 19; i.e., they are still going to use the older alarm API behavior with defaults oriented towards exact alarms. Out of the 22+K apps in the dataset 47.25% use alarms. Of the apps

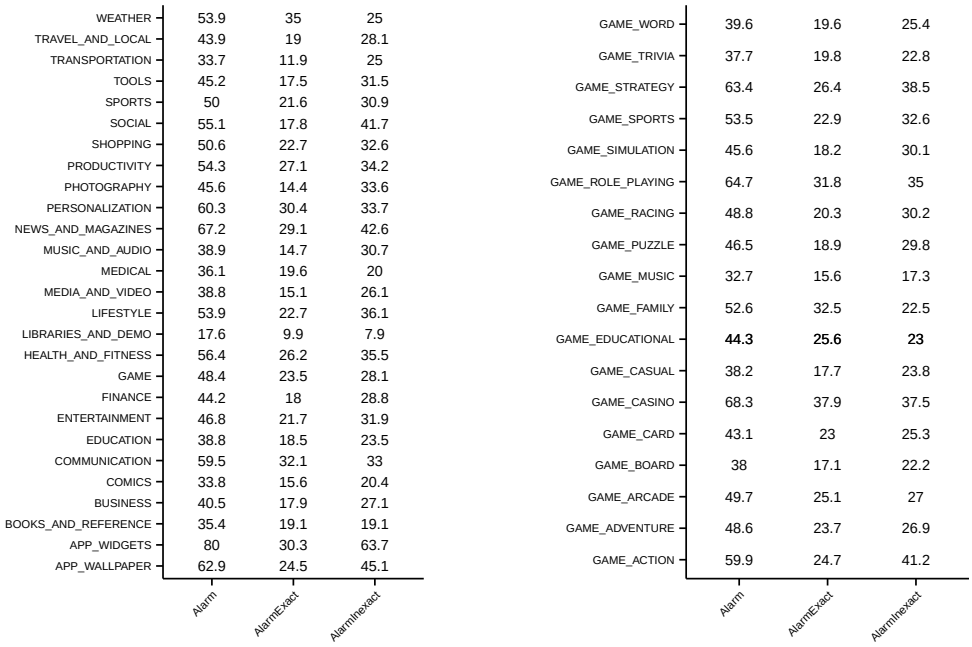
Application	SDK	Downloads	Version
es.lacaixa.mobile.android.newwapicon	17	1M-5M	2.0.17
com.cg.tennis	14	10M-50M	1.6.0
com.linkedin.android	15	10M-50M	3.4.8
com.rovio.angrybirds	18	100M-500M	5.0.2
com.cleanmaster.security	17	100M-500M	2.5.1
com.shazam.android	16	100M-500M	5.3.4
com.instagram.android	16	500M-1000M	6.20.2

Table 4.2 Example of popular and regularly updated apps with more than one million downloads and with target SDK older than 19 months (as of May 2015).

that use alarms, 53.49% have target SDK versions above 19, while 46.06% target older SDKs (Table 4.1). As annotations can affect the targeted APIs on a per method basis, it was confirmed that only 2% of the apps with SDK < 19 had occurrences of the `TargetAPI` annotation in methods containing alarm calls.

The major apparent difference between SDK < 19 apps and SDK ≥ 19 apps is the flip-flop in usage of exact and inexact alarms: only 2.31% of apps targeting SDKs ≥ 19 define exact alarms in contrast to the 44.05% of apps targeting < 19. Note that this change might not necessarily be the result of developers being aware of the impact of exact alarms, but rather an end result of targeting newer SDKs.

The reason behind Android being so conservative with maintaining the previous alarm behavior even in newer versions of Android is to avoid apps from becoming unstable when updating. Since only 2.31% of the apps targeting SDKs ≥ 19 use the exact alarm API call, if one would consider the hypothesis that apps with target SDK ≥ 19 updated from an older SDK, it is probable that either most apps did not have exact time constraints after all or that the ones that do willingly avoided updating their SDKs. If the first is true, then Android is being very conservative with their approach regarding alarms batching behavior, which has a big impact on devices' power consumption. Although this study does not consider apps update rates, it would have been interesting to determine if the second case holds by, for example, determining how many of these apps were updated after the release of Android API level 19. The intuition is that even regularly updated apps often do not update their SDK. As an example, in Table 4.2, a few well known apps are shown which, by the time of this study, had target SDKs lower than 19. Which means that these apps are unable to utilize the new energy efficient alarm APIs provided by the latest Android SDKs.



(a) All categories.

(b) Game apps.

Fig. 4.2 Percentage of apps per category (avg. 523 apps) that have any kind of alarms, have exact alarms and inexact alarms. Due to the high amount of Game categories, a) groups this categories into GAME. Note that an application can make use of both exact and inexact alarms.

Even if the device is supported and up-to-date, apps can target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device. The results clearly demonstrate that there is slow adoption of new SDK versions by application developers. More importantly, one can see that despite the efforts to make Android more energy efficient with respect to alarm handling (e.g., through JobInfo and the introduction of inexact alarms), backwards compatibility (a necessary evil at this point due to fragmentation), lack of developer awareness about new SDK benefits, and misuse of alarms by developers makes it hard to succeed.

4.2.4 Type of Alarms depending on app category

Considering the conservative behavior of Android regarding non-deferrable alarms, one now wonders which type of apps require exact alarms. To this end, this study now explores how different categories (as retrieved from Google Play) of apps make use of alarms (Figure 4.2).

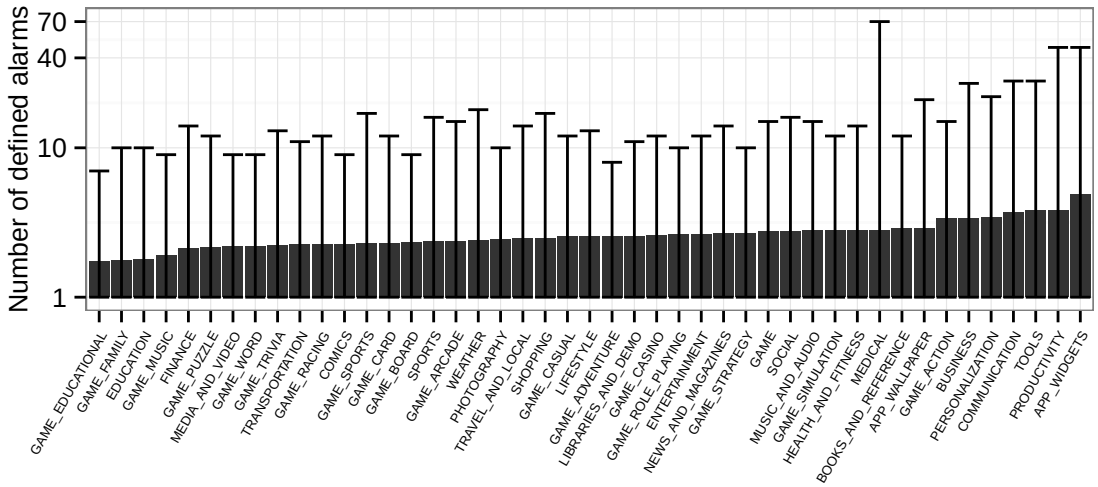


Fig. 4.3 Average number of alarms per application for each Google Play category. Error bars depict the maximum number of alarms for each category.

Surprisingly, categories of apps such as widgets (80%), wallpapers (63%) and personalization (60%) have a bigger fraction of apps with alarms than communication (59%) and social categories (55%). While having more alarm definitions does not necessarily mean that there will be more alarm occurrences during runtime, it was found that, for example, there are 308 widget apps defining repeating alarms (`setRepeating` and `setInexactRepeating`) (in Sec. 4.2.6 some of these apps are manually analyzed). Regarding time critical alarms, the five application categories with most apps with exact alarms are respectively: casino games (37.9%), weather (35%), family games (32.5%), communication (32.1%) and role-playing games (31.8%). Finally, the average number of alarms defined by apps per category is shown in Figure 4.3.

The widgets category not only has the largest number of apps with alarms and one of the highest time critical alarms usage (30.3%), but also it also has the highest average number of alarms (4.9) defined within an application. The analyzed apps had up to 70 alarm definitions[4], e.g., Whatsapp defines 28 alarms, Instagram 11 and Facebook only 2. Again, note that although Facebook has only 2 alarm definitions, its alarms are actually very frequent during runtime (Section 4.2.6).

[4] `com.ecare.android.womenhealthdiary`

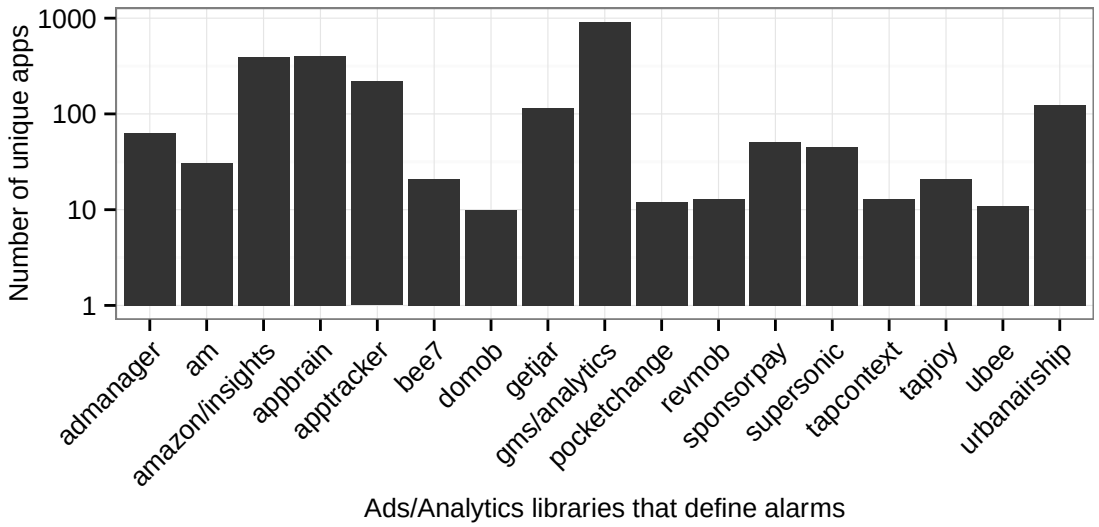


Fig. 4.4 Number of apps with alarms defined by third-party ads/analytic libraries.

4.2.5 The impact of 3rd party libraries

Android apps tend to have a big proportion of 3rd-party content. For example, consider Skype, only about 36.4% of its code is actually Skype-specific functionality, while 31.8% accounts for 3rd-party SDKs (e.g., roboguice, jess, qik, android support) and 32.8% belongs to ads/analytics (e.g., flurry, Microsoft ads).

Hence one important aspect to check is whether defined alarms are native to the application itself or if they originate from 3rd party libraries. To this end, the package names of the files where the alarms were detected were analyzed and compared to 93 ads and analytics libraries available for Android, retrieved from a public list provided by AppBrain[5]. The library package names and matches were manually confirmed to eliminate false positives.

Figure 4.4 shows the number of apps where alarms defined by these ads/analytics libraries were found. Alarms of ads/analytics libraries found in less than 10 apps are omitted (e.g., cellfish, inmobi, mopub). Although this approach might not cover all possible ads/analytics libraries, it was found that at least 10.65% of the unique apps (22.55% of apps with alarms) have alarms defined by third-party ads/analytics, and around 10.42% of all alarm API calls found belong to these libraries. Furthermore, 22.34% of the alarms defined by the third-party libraries are exact.

Finally, considering the number of alarms defined across all apps, it was found that 31.5% of all alarms are repeating, while nearly 40.5% of alarms are non-deferrable. Regarding

[5]http://www.appbrain.com/stats/libraries/ad

3rd-party ads and analytics libraries, their alarms account for 10.4% of all alarm occurrences. From these occurrences, 72.6% of them are repeating and 22.3% of them are non-deferrable. Even though only ads/analytics were explored, given the large coverage of these 3rd-party libraries, optimizing their resource consumption and having them use inexact alarms (e.g., using `TargetAPI` annotation) would certainly lead to appreciable gains in terms of energy consumption.

4.2.6 Occurrence of alarms at execution time

To confirm the impact of alarms on Android KitKat (SDK 19), the first to introduce batching by default, two experiments were performed. The experiments use two different sets of 30 apps. The first set is the top 30 most popular apps of the Google Play market. The second set is the 30 apps with the largest number of `setRepeating` alarm definitions that also target SDK lower than 19. The latter was chosen since these alarms should be deferred if the target SDKs were set to ≥ 19 and notably includes apps with >1K to >500M downloads.

For each experiment a new Android firmware (KitKat) is flashed, install the 30 apps and create new accounts with no contacts/friends when needed (e.g., Gmail, Facebook, Twitter, etc.). All apps were started once to ensure Android gives them permission to execute on reboot if required, and then the phone is left on for around 30 minutes. Then the phone is rebooted, its screen turned off, and left running for around 3 hours. Finally, the alarm and wakeup counts are gathered as reported by Android Dumpsys (`adb shell dumpsys alarm`) for the installed apps. Both experiments were repeated to confirm the observed patterns.

There were a total of 261 alarms registered by the apps in the first experiment. Only 53 (20%) caused the device to wakeup and no significant correlation was found between the number of registered alarms and the number of alarms that woke the device ($r = 0.11$, $p = 0.55$). That said, it came as a surprise that the two Facebook apps (messenger and the regular app) were responsible for the majority of wakeups (15 per hour). Upon closer examination, it was determined that they were waking the phone to maintain a connection to a message queue, even though the accounts used had literally zero social activity.

A total of 1,041 alarms were registered by apps in the second experiment. Of these, 636 (61%) woke up the device and a strong and significant correlation was found between the number of registered alarms and the number of alarms that woke the device ($r = 0.86$, $p < 0.01$). The worst offending application was the social network Spoora (10K-50K downloads) which registers *only* `setRepeating` alarms and also has its SDK target set to 9. Spoora was responsible for 372 wakeups and is a clear example of the negative impact of careless alarm usage which could be easily mitigated by simply targetting a newer SDK.

Interestingly, this type of scenario is not unique to less popular apps: Norton Security and Antivirus (10M-50M downloads) has a target SDK of 17 and caused 141 wakeups.

From these two experiments there is clear evidence that poor alarm API usage can cause substantial impact on the device, and it is not limited to small time developers. In particular, these results highlight how even a simple misconfiguration (i.e., setting a target SDK too low) can have significant negative impacts in execution behavior. A future direction would be to perform similar experiments on a larger scale, taking direct battery measurements, manually modifying the target SDK to quantify exactly how the impact on battery consumption changes between target SDKs, and more closely examining the relationship between alarm type declaration and registration/wakeups.

4.3 Conclusion

Research on energy efficiency in mobile devices tends to propose solutions focused on batching activity to amortize the cost of waking up the mobile device and its radio. The efficiency of such solutions depends on the ability of the operating system to schedule background activity at the most appropriate time. In Android, alarms are a popular mechanism to schedule background activities. To understand apps' usage of alarms, over 22 thousand of the most popular apps in the Google Play store were crawled and downloaded.

It was found that nearly 50% of apps define their alarms to be *non-deferrable* by the operating system, thus hamstringing Android's ability to optimize scheduling at all. When examining the prevalence of alarms, it was found that they exist across all categories of apps with some having up to 70 alarms declared. For apps with alarms, 22.5% have them defined by 3rd party ads/analytics libraries they use, and these libraries account for at least 10.4% of all declared alarms. The inefficiencies of alarms were also shown by manually analyzing 60 apps at runtime, finding apps waking up the device an inordinate number of times.

While Android fragmentation has been studied in the past [77, 106, 110], it was generally approached from the perspective of the wide distribution of Android versions, heterogeneous hardware, and lack of updates. In this work a new facet of this problem was revealed: even if the device is supported and up-to-date, apps often target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device. Via static analysis, it was discovered that a substantial number of apps' alarms are non-deferrable due to targeting older versions of the Android SDK and that by simply changing the target SDK to > 19 these apps would likely benefit from advanced OS alarm scheduling mechanisms. Further, while previous work [106] which studied a much smaller set of 10 open-source apps found that 28% of method calls were outdated with a median lag time of 16 months, this

study shows that in the case of alarms, close to half the API calls are outdated by more than 18 months.

Ads and analytics are a particularly interesting subject of study since they have been shown to have a big impact on energy consumption [76]. In this dataset the majority of alarms related to ads and analytics are repeating, meaning that they most likely result in background operations that might have no real end-user benefit. This seems to be a problem that is core to Android in particular, since iOS does not have a direct analogue to alarms and has an extremely limited background execution environment [25]. Since from the experience a large proportion of Android apps make use of third-party code, future large-scale studies of energy consumption, optimization, and alarm usage should focus on common third-party libraries.

When examining alarm usage at runtime, the implications of the static analysis held true for the most part. The apps with the highest number of defined alarms were in fact executing the alarms at an exceedingly high rate. In one egregious case, a single application was responsible for 372 wakeups in a 3 hour period.

This work serves as an initial large-scale look into alarms and their impact. Overall, the findings indicate that research on energy efficiency on mobile devices needs to incorporate an understanding around the use of alarms. Deeper examinations into the use and abuse of Android alarms should provide more fruitful insight and solutions, leading to increased energy efficiency and device performance.

We have seen that the Android ecosystem is complex and the interaction between the multiple entities and developer mistakes can result in performance degradation. While some of these mistakes can be resolved through better developer awareness, others require intervention at the OS level. While static analysis provides a decent indicator for detecting potential inefficiencies, their runtime impact can vary. Therefore it is important to complement such information with runtime dynamic analysis data (as seen in Section 4.2.6). Nonetheless, dynamically analyzing closed-source apps either at large scale or with realistic inputs is still challenging. The next chapter discusses and addresses some of these challenges.

4.3.1 Results Summary

Some of the findings of the Chapter 4 include:

- Close to half of the apps are making use of execution scheduling APIs (Alarms).
- Close to half of the most popular apps performing execution scheduling (via alarms) do not benefit from energy efficient batching capabilities.

- For apps with alarms, 22.5% have them defined by 3rd party ads/analytics libraries they use, and these libraries account for at least 10.4% of all declared alarms.
- Through analyzing 60 apps at runtime it is shown that OS fragmentation and the misuse of alarms can greatly increase the number of times a device wakes (up to 12 times more often).
- Android SDKs tend to lag behind by more than 18 months.

Chapter 5

Large Scale Dynamic Analysis

Developing mobile apps is becoming easier by the day but testing apps is still hard. The de facto standard testing tool, called "Monkey," is based on random inputs and performs well on simple (and sometimes complex) apps, but fails in gathering input like user engagement and attention. For this reason, app testing still benefits from user inputs gathered by distributing apps, or even phones with pre-installed apps, to real users. This approach is quite expensive and has an intrinsic scalability limitation.

In this chapter CHIMP is proposed, a system that challenges the state of the art in app testing and provides large-scale human inputs. The key idea behind CHIMP is to run mobile apps in a virtualized environment while streaming their content to users' browsers. Behind the scenes CHIMP collects data like user input, app performance, and app network traffic. CHIMP's design details are described and its efficacy is demonstrated by testing hundreds of apps via thousands of crowdsourced users. CHIMP outperforms the "Monkey" in most app categories, improving code coverage (up to 25%) and the generated traffic volume (up to 3x). Finally CHIMP's potential is demonstrated by building a traffic classifier on encrypted network flows, achieving f1 scores of above 0.9.

5.1 Introduction

The popularity of mobile apps is no longer in question: according to recent estimates, mobile traffic is expected to grow nearly 8 times between 2015 and 2020 [42]. Such an explosion is heavily driven by the increasing amount of tools and libraries that facilitate app development. As an example, the Apple Store currently receives over 1,000 new apps per day [140], and app downloads are expected to increase four times by 2020 [22].

While developing apps has become easier, testing them remains challenging. This is because of a significant lack of tools for large scale testing and measurement of mobile apps.

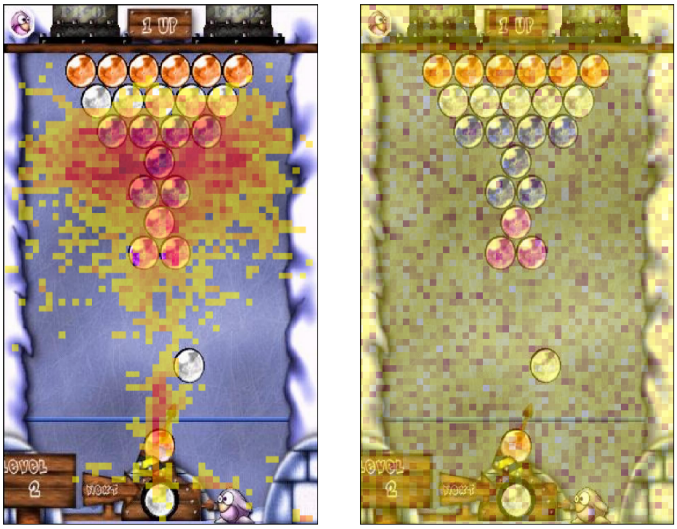


Fig. 5.1 Human input (100 users) vs monkeys when playing frozen-bubbles.

The de facto standard app testing technique is to use "monkeys" [40]. A monkey is a simple tool that performs random (partially configurable) inputs, operating under the assumption that a million monkeys tapping on a million touch screens will eventually expose faulty code.

While this might be true, there are several issues. First, prior work has shown that monkeys are not well suited for certain types of inputs [40], e.g., filling out forms. Second, monkeys' inputs do not reflect those of actual app users. Figure 5.1 visually shows this issue when testing "frozen bubble", an Android game, via both real users and monkeys. While real users focus on the part of the screen where game play actually happens, monkeys make no distinction and spread their efforts across the entire UI. While this might have some advantages if exploring all code paths is the desired outcome, it is very much a problem when looking for, e.g., the way an apps' users access the network, understanding how users will navigate through options/menus, or evaluating the impact of a change in functionality or UI.

This is the reason that mobile app testing does not yet live (entirely) on the planet of the apes: humans are still needed to test applications, both in industry and research. Unfortunately, while large-scale human testing is (mostly) achievable for giants of industry (e.g., Apple, Google, Facebook, Microsoft, and Amazon), many smaller developers do not have thousands of users to A/B test with, or control over app delivery mechanisms (i.e., app stores) [17, 66]. Indeed, even solutions proposed by app store operators have their own limitations.[1] Further, the research literature is littered with examples where authors of papers spend many hours manually running apps (i.e., performing dynamic analysis) to better understand a variety of issues related to mobile apps [96, 117, 127].

[1] E.g., In [66] only owned apps can be tested and users must install them.

In this chapter CHIMP is proposed, a flexible Android app testing system that enables quick collection of human inputs for mobile apps. CHIMP runs apps on a server and streams them to a browser for real users to interact with. While users test apps, CHIMP collects a wide range of data (user interactions, network traffic, runtime traces, etc.) as well as explicit user feedback. "Experimenters" (e.g., app developers or researchers) can use CHIMP with the apps they would like to test and specify the data they would like to collect via *campaigns*. CHIMP offers integration with CrowdFlower [44], along with user validation techniques, to quickly provide large, trust-worthy datasets. For example, the human input behind Figure 5.1 was obtained from 100 CrowdFlower users in just a couple of hours.

First, the design (§5.2) and evaluation (§5.3) of CHIMP is presented with thousands of apps and users. By analyzing user interactions more useful campaigns (§5.4) were designed. In (§5.5 and §5.6) it is shown that integrating users in the testing loop can improve code coverage by up to 25% and generate up to three times more traffic volume. Using CHIMP, it was possible to gather the data necessary to train and test an app traffic classification model, achieving f1 scores of over 0.9 in some cases. Finally, the implications of the findings are discussed and a promising direction for future development of automated testing tools in §5.7 is presented before concluding.

5.2 Design

CHIMP is made available as a web application, and users interact with it via the *web-client*. Within the web-client, they can interact with an Android *virtual phone*, where mouse clicks and drags get translated into taps and swipes.

CHIMP integrates several technologies behind the scenes to achieve its goal of providing insights on mobile apps via both objective measurements (e.g., application runtime analysis, network traffic, permissions) and feedback from users.

Further, to meet many of its goals CHIMP must scale reasonably well and be flexible enough to support different types of analysis, number of users, and apps. This necessitates addressing a few challenges that are discussed in this section: 1) web-client workflow, i.e., how to organize and present tests to users, 2) selecting an effective means of virtualizing a phone for users, 3) supporting a multi-user experimentation platform, and 4) collecting useful data for experimenters.

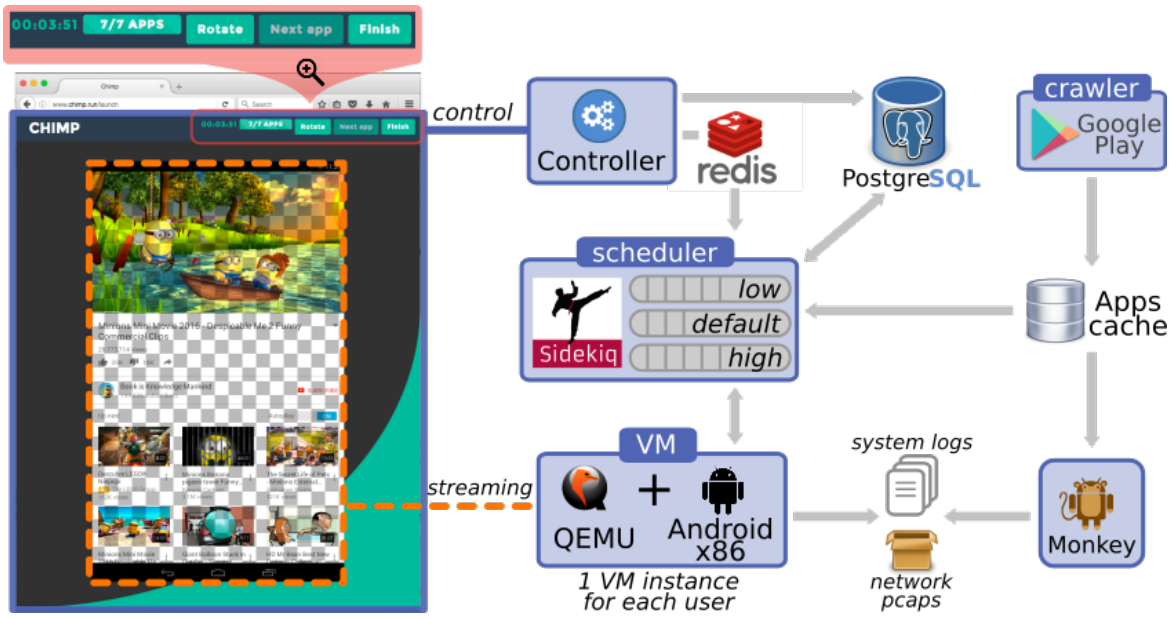


Fig. 5.2 CHIMP's system composition.

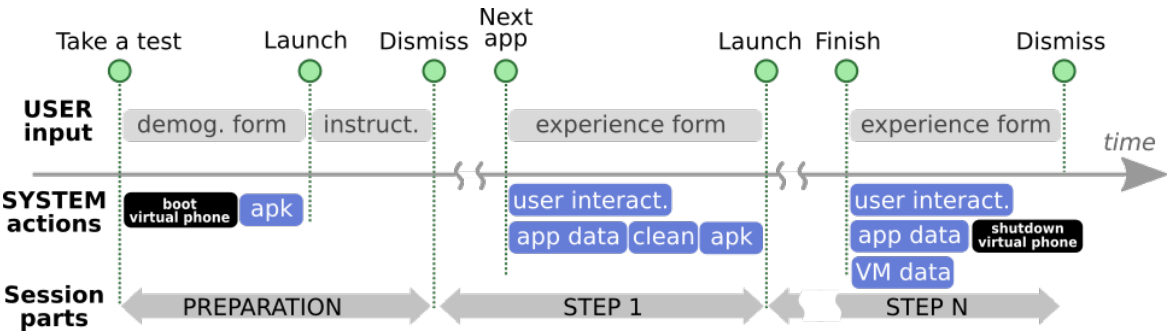


Fig. 5.3 CHIMP's timeline detailing the progress of a user session.

5.2.1 Web-client Workflow

Figure 5.2 sketches CHIMP's architecture. As an illustrative example, the figure includes a screenshot of what a user sees when interacting with the YouTube app. In particular, the webpage presented to the user is composed of a *streaming area* replicating the content of the virtual phone's display (dashed area in the figure), and a *control area* allowing the user to issue specific commands to the system (zoomed area in the figure).

There are a number of actions that must be taken before the user can actually interact with an app, which are illustrated in Figure 5.3 with a timeline, along with the associated system actions (bottom). In the following, this visual aid is used to describe the status of the user's interactions with CHIMP.

A *session* is defined as the entire set of actions a user takes in CHIMP. A session starts when the user visits CHIMP's homepage and presses the `<Take a test>` button. A welcome message is presented, including a form to collect demographic data. While the user is busy filling out the form, CHIMP prepares a virtual phone (§5.2.2) and installs the first app. When the virtualized environment is ready, users can press `<Launch>`. They are then presented with a set of instructions on how to use CHIMP. Once the instructions are dismissed, users can interact with their first app.

Since CHIMP allows users to interact with different apps, each session is partitioned into multiple *steps*; one per app. The control area lets users navigate through steps via the `<Next app>` and `<Finish>` buttons. Specifically, users interact with the current app until clicking one of these two buttons. After clicking one of these two buttons, an *experience feedback form* with a set of questions is presented.

If the user pressed `<Next app>`, the previous app is removed from the virtual phone and a new app is installed. After filling out the experience feedback form, users can interact with the new app. If instead the user opted for `<Finish>`, the virtual phone is shut down and the session terminates after the app experience feedback form is completed.[2] The control area also shows a cumulative session duration and overall step progress (∼4 minutes and 7 apps in Figure 5.2).

Multiple user sessions with the same setup are logically grouped into a *campaign*; more details on campaign customization will be discussed in §5.2.3.

5.2.2 Android Virtual Phone

The core of CHIMP is built around the virtualization of (Android) mobile devices. This is accomplished by instrumenting virtual machines (VM) running the Android operating system to provide the user with an Android Virtual Phone (AVP). To maximize app compatibility, CHIMP should be able to 1) execute ARM instructions (to support apps that use native binaries that target it), 2) support OpenGL (especially for games), and 3) offer fluid interactivity. Several existing solutions were evaluated and thus the respective lessons learned are now discussed.

Android VM: The first obvious solution is the Android emulator (Emu) which comes in two flavors: Emu-ARM and Emu-x86. Emu-ARM refers to the original Android emulator which implemented the ARM instruction set in software. Emu-ARM is known to suffer huge performance penalties when running on x86 architectures [85], and so is often replaced

[2] Users *can* continue past the feedback form without actually filling it in.

by Emu-x86. Notice that, despite the name, Emu-x86 is actually a VM hosting a build of Android targeting the x86 instruction set.

While Emu-x86 speeds up app execution, it introduces the problem of translating instructions for native binaries that target ARM. Android apps are mostly built in Java which (in theory) does not target specific hardware at all, but they *can* make use of native code, either via libraries or even portions of the app itself. Since ARM dominates the mobile landscape, most IDEs (and build systems) compile ARM for native code by default, and leave x86 support as optional. The real-world implication of this is that native x86 binaries cannot be assumed present in apps. This means that while Emu-x86 is a good solution when *developing* apps, it does not fit the needs.

To deal with running ARM code on x86 chipsets, Intel developed *houdini* which performs on the fly translation of ARM to x86 instructions. Fortunately, the community-driven port of Android for x86 architectures (Android-x86 [20]) has native support for houdini [21]. It was opted to use QEMU directly, a popular open source hardware emulator and virtualizer that Emu-x86 itself uses behind the scenes. Using QEMU directly gives fine-grained control over VM RAM allocation, CPU core usage, and supports output streaming via websockets.

While QEMU takes care of most hardware virtualization tasks (using Kernel Virtual Machines), particular attention is needed to deal with OpenGL. QEMU supports *virglrenderer*, a virtual GPU that provides the Android guest VM with access to the host's GPU for hardware accelerated graphics (i.e., OpenGL support), however it is not usable out of the box. Integrating virglrenderer required recompiling the Android-x86 image and QEMU to enable GTK library support. Although this was closer to the optimal solution, integration was not perfect: mouse input under GTk 3.16 was still creating some issues, and thus CHIMP only has partial support for OpenGL at this time[3]. Next, since QEMU emulates a VESA-compliant VGA output device, the VGA BIOS was modified to support WVGA resolutions that are found in mobile devices (e.g., 400x800).

As reported in Figure 5.2, each user session is associated with a separate QEMU/Android-x86 VM, which corresponds to one *virtual phone*. Overall, CHIMP's virtualization is composed of two technologies: 1) Android-x86 to execute the Android operating system and apps and 2) QEMU to virtualize the phone hardware.

Streaming: QEMU natively supports Spice [137] and VNC (an implementation of RFB [128]), two popular streaming technologies. Additionally, streaming can also be done via XSpice [138], a variant of Spice which uses an X11 server. All three options were tested (results not reported for brevity), by creating an HTML5 client handling the streaming from the virtual

[3]The virglrenderer integration is an ongoing effort by the Android-X86 and Qemu communities and many of the issues faced in this thesis were already fixed in later versions of Android X86.

phone, i.e., the streaming area in Figure 5.2. In the end, VNC was selected since it offered a more fluid experience throughout the tests.

5.2.3 Experimentation Platform

While the AVP is a necessary component for implementing CHIMP, it is not sufficient. CHIMP must allow experimenters (i.e., researchers or app developers) to specify campaigns and dynamically launch and manage AVPs for users.

Campaigns: As previously mentioned, user sessions are grouped into campaigns. A campaign is a set of parameters that includes the target number of users, the set of apps to test, how many apps per user, if apps should be presented to users in a random or pre-set order, the amount of time to be spent on each app, and what data to collect (§5.2.4). The instructions for the campaign and interstitial feedback forms are also customizable via `JSON`.

Orchestration: The different components of CHIMP are orchestrated via a *controller* and *scheduler* (see Figure 5.2). The controller tracks events issued by the web-client control area, and triggers *job* scheduling. E.g., when clicking the `<Next app>` button the controller schedules jobs to retrieve data from the virtual phone and the web-client, and to prepare the virtual phone for the next app (if any). The controller is a multi-process, multi-threaded, and stateless Ruby on Rails application served from Puma [122] webservers which run behind an Nginx [111] reverse proxy on the server.

Sidekiq [136] was used as a job processing framework, which in turn uses Redis to back job queues. As pictured in Figure 5.2, CHIMP's scheduler makes use of 3 queues with priorities *high*, *default*, and *low*. The high priority queue is reserved for system critical jobs that have tight scheduling deadlines (e.g., extracting code coverage metrics over time). The default queue handles jobs related to user experience, e.g., virtual phone booting/shutdown and app installation. Low priority jobs handle things like reclaiming resources from timed-out sessions and post-processing of campaigns for reporting purposes.

Jobs themselves interact with AVPs via the Android Debug Bridge (*ADB*), and are often chained together to perform more complex operations. E.g., booting a virtual phone is done by chaining four jobs: 1) replicating an Android image and booting it via QEMU, 2) making the VM accessible to the user (i.e., opening ports and configuring mappings), 3) enabling measurements (i.e., setting up communications between the AVP and requested data collection modules), and 4) scheduling an app installation.

5.2.4 Data Collection Modules

CHIMP allows the collection of data via different modules. Since CHIMP collects data from human participants[4] it was assured that: 1) only the minimum data required for the system to achieve its goals was collected, 2) any sensitive data was anonymized, and 3) users were informed of which data was collected prior to each session. Although there is the intention to expand the type of data collected in the future, for now CHIMP supports collecting six types of data. Now both *what* data each module collects and *how* it is collected are described.

User interactions: The web-client collects mouse events from the streaming area, as well as information on browser focus.[5] This is achieved by instrumenting the web-client with JavaScript attached to the HTML5 VNC streaming area. This data is crucial to understand user behavior (e.g., identifying where users click as in Figure 5.1), to validate the quality of the crowdsourced workers (§5.4), or to reproduce crashes by replaying inputs. The web-client uploads user interaction data to the server at the end of each session step (Figure 5.2).

User feedback: While the virtual phone is booting, the users are asked to provide some demographic feedback (age, gender, country, and mobile app expertise). Additionally, at the end of each session step users are asked to provide some feedback on their experience using the app, i.e., if the app crashed or required login, give a rating on "fun" and "speed" of the app (on a scale of 1 to 3), and place the app in one of several pre-defined categories (e.g., social, multimedia, or game).

App data: At the end of each session step, the execution history of the app (`logcat` on the virtual phone) is retrieved, to identify exceptions or crashes (if any). Similarly, `dumpsys` is used to collect app's resource consumption (CPU, memory, disk I/O, network), interactions with the operating system, permissions, sensor usage, etc. App meta-data (e.g., number of downloads, category) as well as app structure (e.g., classes, methods) are retrieved via static analysis of apps downloaded from the Google Play store.

Runtime data: The app runtime execution (§5.5) is captured via method tracing. Additionally, EMMA [52], the defacto standard of code coverage analysis, is also supported for opensource apps. It also allows running automated monkeys whose inputs can be used in conjunction with humans.

Network data: the network traffic originated by the app is collected in two ways. First, *tcpdump* is ran to collect raw pcap files. Second, the Android `/proc/net` polled to obtain the list of open network sockets along with their UID[6]. This solution requires some calibration to be able to capture ephemeral flows (i.e., very short lived connections). In the tests, it was

[4] All data was collected in compliance with the institutional ethics guidelines.
[5] No activity is monitored on other browser tabs nor raw keyboard inputs.
[6] NB: in Android each app is given a different numerical user id (UID).

found that only 1% of the flows were shorter than 100 ms, so a 50 ms polling frequency as considered as a conservative choice. This activity runs in background in the virtual phone. Furthermore, the UID was mapped to the app's package name using the information provided by `dumpsys`.

System data: A module that monitors both the whole system health, as well as the connectivity toward each individual user web-client and the associated virtual phone. In particular, the web-client runs HTTP pings every 5s towards CHIMP's webserver,[7] and collects frame buffer updates from the streaming area. The system records also the current workload (e.g., virtual phones running, memory available, etc.), as well as the time-line progress of each individual user session using a combination of `collectd` [43] for collection of system metrics and `graphana` [74] for visualization.

5.3 Implementation

This section describes the details of CHIMP's implementation. It starts by briefly reporting on the alternative choices that were contemplated. Next, the prototype is evaluated in terms of resource consumption and user experience. Finally, its limitations are discussed.

Setup: CHIMP was deployed on a server with a Dual Intel Xeon CPU E5-2697v3 (2.60GHz), with 128 GB of RAM, and a single 7,200 RPM hard disk with 130MB/s write throughput. To control the Android VMs the default ADB implementation was used with automated device detection, which is limited to a predefined range of only 15 ports. While this could be trivially extended by having CHIMP managing the adb connections, CHIMP aims to scale horizontally by adding smaller on-demand backend machines and so the following evaluation operates within this limit.

For Android VMs a customized Android-x86 image (4.4-RC1[8]) was used, which requires around 1.6 GB after installing when using QEMU's *qcow2* image format. Qcow2 provides support for snapshots (i.e., saving/resuming a VM state), which will be used in the future to speed up VM booting. A copy of the image is made for each user; two approaches were tested to store these images, i.e., in disk and in volatile memory, i.e., a *ramdisk*. For both setups each of QEMU's caching policies was benchmarked, but in the following only the two best performing strategies are compared: writeback caching when using disk, and no caching when storing images in RAM (using tmpfs, a RAM-based file system).

To evaluate CHIMP's implementation, the top 500 apps per category in the Google Play store were crawled, retrieving a total of 18,787 unique apps. For users, both "synthetic" and

[7] It is not possible to run ICMP pings in JavaScript.

[8] The latest supported version is 6.0.

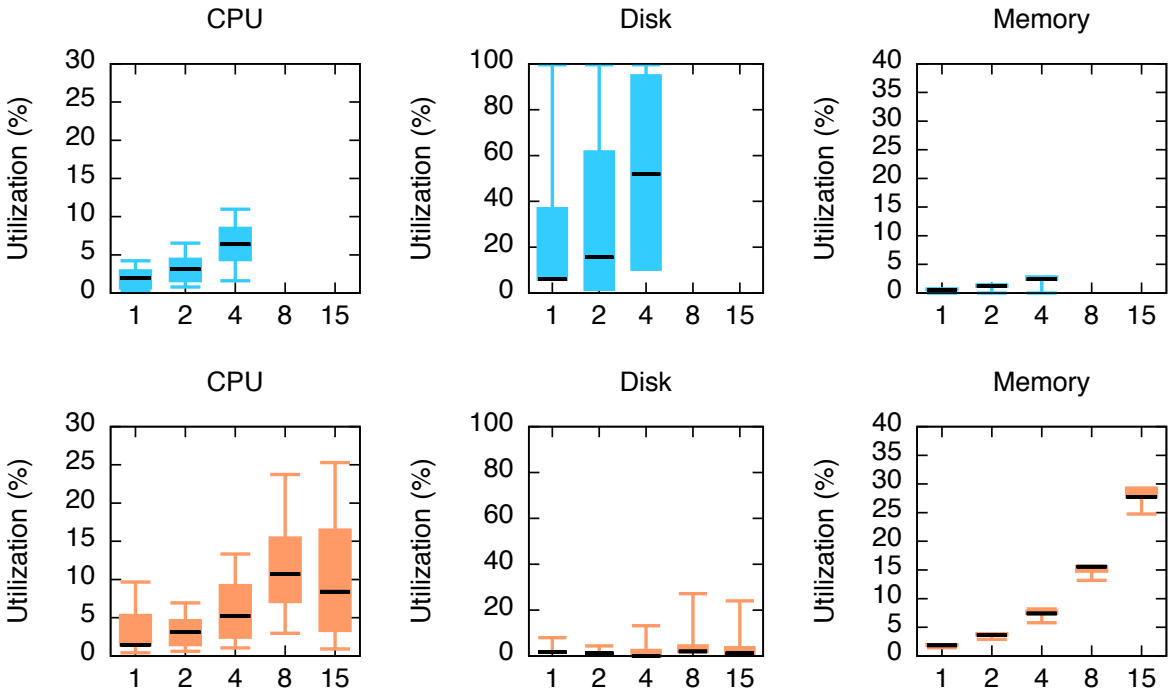


Fig. 5.4 CPU, disk, and RAM utilization as a function of number of concurrent users. Rows are HDD and RAM based disks, respectively.

human users were used. Synthetic users were created with *Selenium* [134], a framework for automating web browser interaction, while 1K real users were recruited on CrowdFlower. Synthetic users were leveraged to cover the full set of apps (at no cost) while stressing the implementation, i.e., up to 15 simultaneous synthetic users were launched. Real users were leveraged to test a subset of apps (1K randomly selected, non-crashing apps) and to collect explicit feedback on system performance. A summary of these two campaigns appears in Table 5.1 as "Automation" and "Discovery," respectively.

App Compatibility: The first major question to answer is simply how many *real* apps does CHIMP support? Using the app data collection module it was found that while 13,374 of the apps in the campaign installed fine, 474 failed to install, and 4,939 failed to be brought to the foreground of the AVP (i.e., the app did not successfully launch). Failing to install can be an indication that the app binary is broken, incompatible with hardware/drivers (e.g., full OpenGL support in CHIMP is still a work in progress), or it takes too long to install (an upper bound of 40s for app installation was enforced). Failing to be brought to the foreground is often a sign of a crashing app. Regardless, 71% of apps were installable and successfully

brought to the foreground, indicating the CHIMP has pretty good support for real-world apps.[9]

Resource utilization: Next, to get an idea of how the design choices play out in terms of resource usage, Figure 5.4 plots the distribution of CPU, disk, and RAM utilization as recorded by CHIMP during the synthetic user campaign. Notice that for disk, the utilization corresponds to throughput of the disk. The server had plenty of CPU available to sustain the workload, e.g., in the worst case (15 concurrent users), a maximum of 30% CPU utilization was observed. Instead, CHIMP is limited by disk I/O, e.g., up to 100% disk utilization (130MB/s) with only 4 concurrent VMs and a physical disk.[10] Using tmpfs alleviates this contention and allows CHIMP to seemingly scale up to 15 concurrent VMs.[11] In this case, RAM usage is linear since resource consumption is driven by the size of Android images (previously on disk) and the RAM allocated to the VM itself. A maximum of 10 and 7 Mbps, network transmission and reception rates were observed, respectively.

Based on the analysis above, CHIMP uses tmpfs and the controller (§5.2.3) ensures that system load remains below configurable thresholds. This means that new users might be put on hold when attempting to start a new session if not enough resources are available. Such functionality is key when regulating the user load from CrowdFlower. CrowdFlower does not provide APIs to request a user arrival rate, however, its jobs can be started, paused, and stopped at the job owner's discretion. The CHIMP controller was engineered to take advantage of this "trick" to regulate the system workload.

Service latency: To benchmark CHIMP's performance the boot and app install times were collected for the synthetic users using tmpfs. To summarize, on average, when a user starts a new session he faces a waiting time of about 36 seconds: 21 seconds to boot (at least 12s less than from disk) and on average 15 seconds to install and launch apps. While the boot time is stable, installing an app instead varies with the app size, taking between 10 seconds for small apps (a few hundred KB) to 35 second for large apps (hundreds of MB).

User experience: Here, a couple things are investigated. First, how "fast" app interaction felt to users. Next, if real users were able to trigger any crashes that the synthetic users did not detect. Finally, how many users had previously used each app, how fun they thought each app was, and whether or not they saw ads within the app.

Figure 5.5 plots the Cumulative Distribution Function (CDF) of the mean question scores reported by users, per app, for the 1,000 Google Play Store apps tested. Binary questions (i.e., yes/no) are on the left while questions scored on a scale of 1 to 3 are on the

[9] In comparison, Google's official mechanism [71] to execute Android apps in the browser supports 58.7% of the tested apps [72].

[10] Higher workloads were not tested when loading images from disk.

[11] Similar results were achieved with SSD and loading temporary snapshots with copy-on-write.

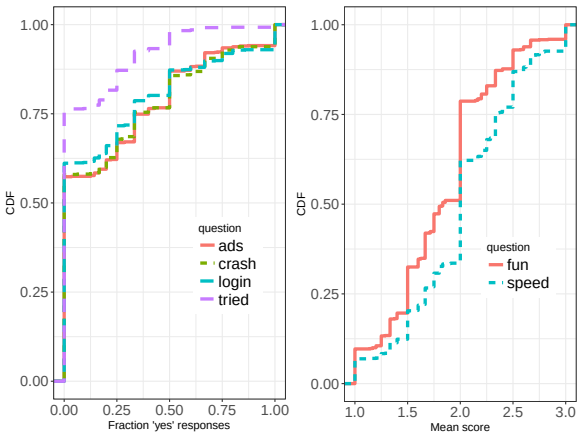


Fig. 5.5 CDF of mean question scores per app for 1,000 tested apps.

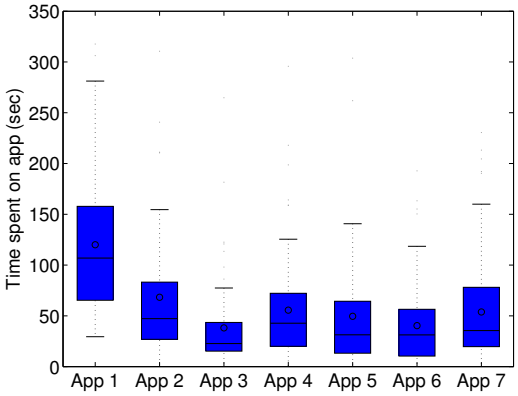


Fig. 5.6 Boxplots of time spent on each app.

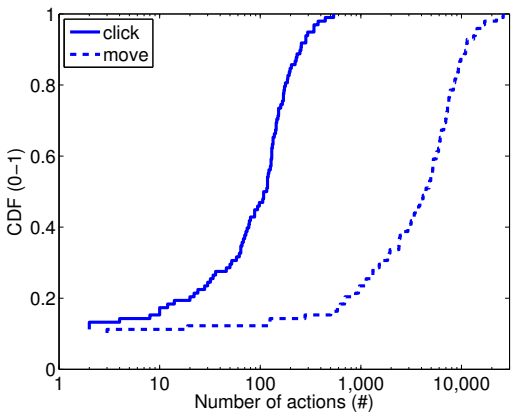


Fig. 5.7 CDF of actions (clicks and move) per user.

right. From the left plot, one can see that most of the apps were new to the users (90% of apps tested had a mean "tried before" score less than 0.5), that most users reported most apps as *not* having ads/requiring a login. While no crashes are reported for most of the apps, about 15% of apps have a majority of users claiming crashes. From the plot on the right, one can see that users found the apps a bit boring with about 50% receiving a mean score less than 2.0. Regardless, users felt that CHIMP was providing at least a "decent" app experience: over 60% of apps receive at least a 2.0 mean score with respect to speed. Unfortunately, no conclusive correlation could be reached between system metrics and user's reported app experience. This is an articulated subject since multiple factors need to be considered including users connectivity, apps characteristics (e.g., unresponsive apps due to bad design, resource consumption, and/or CDN content policy retrieval), etc. A more careful characterization of these effects is left for future work.

Limitations: While CHIMP does support many of the top apps on the Google Play Store, despite its flexible design, there are some limitations. The most obvious is due to the lack of a physical mobile device which limits testing with respect to sensors, mobility, debugging for specific device types or multi-gesture touches. Additionally, its virtualization targets Android specifically, and does not support iOS or WindowsMobile. Next, there are some apps for which CHIMP is a sub-par platform. For example, certain types of apps (e.g., background services or boring apps) might not stimulate users enough for them to provide meaningful interactions. Second, there are many apps whose usage requires an account or only becomes useful with some sort of network effect (e.g., Facebook). Although accounts could have been created that were automatically included in AVPs, not only it would require manual work, but it might also violate service provider terms of services, while not necessarily representing reality. That said, users are expressly instructed to never enter personal information into apps.

Finally, there are some types of experimentation that CHIMP is just not well suited for. For example, understanding mobile network conditions is better left to tools like Mobilyzer [113]. Additionally, certain experiments may be more longitudinal in nature, while CHIMP's AVPs are currently bound to a session that will eventually time out.

5.4 Campaign Calibration

Apart from the engineering challenge to build CHIMP, a more fundamental challenge lies in dealing with real people. To collect meaningful data, one needs to engineer campaigns that people will be happy to take, e.g., select the right amount of apps per session. One also need to validate user input collected to avoid random clickers or distracted users.

	# Users	Duration	User pay	# Apps	# Steps
Automation	-	1 day	-	18,010	100
Discovery	1000	1.25 days	$120	1,000	7
Calibration	100	3 hour	$12	7	7
Code Coverage	1000	1.4 days	$120	59	4
Trace Coverage	500	1 day	$60	55	4
Traffic Classification	500	22 hours	$60	75	4

Table 5.1 Summary of CHIMP's campaigns. NB: CrowdFlower charges an additional 20% processing fee.

In this section, CHIMP is used to get insights on how campaigns should be structured. Accordingly, "calibration" campaign (Table 5.1) is ran with sessions containing 7 steps, i.e., 6 apps plus a "control" one (see §5.4.2 for details), presented in random order. The tested apps were: 1) adobe sketchbook, an app to draw and paint images, 2) divideandconquer (an open source game), 3) frozenbubble (an open source game), 4) pou (a popular, closed source game), 5) youtube, the most popular app to watch videos, and 6) buzzfeed, a news aggregator app. These apps were chosen relatively arbitrarily with the goal of having a fairly diverse and popular set of apps.

The calibration campaign consists of 100 CrowdFlower users. Only "historically trustworthy" users are requested which comes at the cost of longer recruitment time (3 hours at a cost of $12). Users exhibit roughly a 75/25% male/female gender split, and they are located in 30 countries (Venezuela being the most popular). No minimum is enforced for the number of steps that users should take, leaving them free to skip steps or even finish without having tested all 7 apps. Similarly, no minimum is enforced for the time a user should spend on a given step, the goal being to estimate how much work each user is willing to do "naturally."

In the remainder of this section, first the user behavior is investigated while interacting with CHIMP's website, with the goal of identifying guidelines for future measurement campaigns (§5.4.1). Then, various techniques are compared for discarding unreliable responses (§5.4.2).

5.4.1 User Behavior

First, the users navigation through a session is investigated. Although not shown due to space limitations, 80% of the users interact with all 7 apps and only 7% of the users interact with a single app. Regardless of their progress in a session, 100% of the users click the `<Finish>` button and redeem their completion code (required for payment). It is worth reporting that

these (high) utilization numbers were reached through various iterations on the website's design. Specifically, the addition of a progress bar (Figure 5.2) generated an impressive 40% improvement.

Next, the time users spend per app (Figure 5.6) is investigated. Note that any time the user spent interacting with a other browser tabs is subtracted. Users tend to spend a decreasing amount of time on subsequent apps, e.g., the median decreases from 105s (first app) down to 30s (last 3 apps). The third app is not an exception to this trend but rather an artifact of the methodology. While regular apps are placed randomly, it was opted to show a control app (§5.4.2) in the middle of the session to increase the chance of users testing it. The control app is very simple, it is thus realistic that users spend a little time on it.

Finally, the users interaction with apps via mouse movements and clicks is measured. Figure 5.7 shows the CDF of the number of mouse clicks/movements per user restricted to the virtual phone area. Overall, the figure shows about 10% of "inactive" users, i.e., users with neither clicks nor movements. These users, as well as users with very few clicks, quickly navigate through CHIMP to redeem a payment, e.g., none of them test the 7 apps.

Based on the analysis above, it was decided to structure future campaigns with four apps (three plus control). The rationale is twofold. First, maximize the number of users who will complete a full session; second, increase the chance that user will spend useful time on an app, without just quickly skimming through it. Apps are also randomly associated to steps to make sure they get comparable amount of user time.

5.4.2 Crowdsourcing and Response Validation

No standardized methodology exists to determine the quality of a crowdsourced user. CHIMP leverages CrowdFlower's help to select "historically trustworthy" users. In addition, inspiration is drawn from the validation methodology proposed in [148] as it dealt with similar issues (§2.2).

Specifically, CHIMP leverages "engagement" as an indication of user quality. Engagement is defined by the amount of mouse clicks and movements detected in the virtual phone area (Figure 5.7). To avoid setting arbitrary thresholds, users who never interact with the phone area are discarded, i.e., about 10% of the users according to Figure 5.7. Clearly, users with a single click during a session should be discarded as well but additional filtering techniques are leveraged to better capture such users.

Furthermore, *control questions* are also used, i.e., questions to which the answer is known a priori [80], to identify low quality users. An Android app was implemented that simply asks participants to press three numbered buttons in either ascending or descending order. Failure to produce any of the requested order, or to even press a button, is considered as

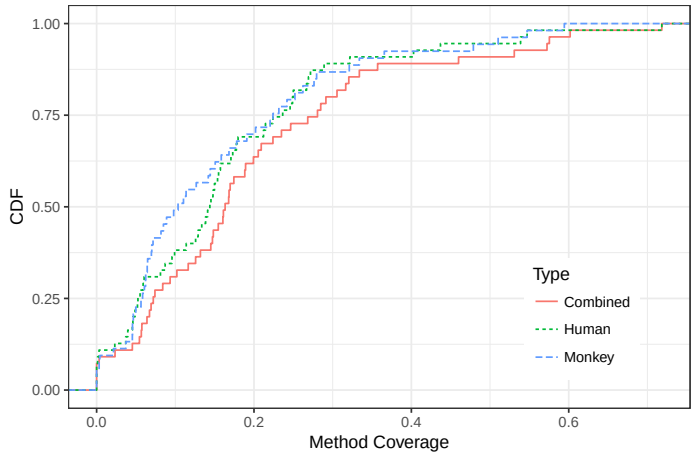


Fig. 5.8 CDF of method coverage achieved by humans, monkeys, and CHIMP's combination of both.

a sign of a low quality user. The control is implemented as an Android app, rather than a simple question within the interstitial forms, since it is a bit more realistic. In fact, users need to interact with it in the same way as they do with other apps and they might face a similar frustration due to app loading time.

Only 3% of the users *fail* the control, which indicates the control was well designed; however, 22% of users *skip* the control app. Users failing the control are labeled as low quality and discarded (these users are still payed). Users that skipped the control are given a second chance for redemption: they are not discarded if they show high level of engagement with other apps. Specifically, such users are required to have at least 100 clicks across the entire test session. This filtering rule introduces an additional 11% of users that are labeled as low quality, and fully captures low engagement users discussed above. The remaining campaigns used in this chapter exhibited the same rate of attrition.

To summarize, a "control" Android app of own design is used to identify low quality users within a CHIMP's campaign. Engagement is also used, i.e., a minimum of 1 mouse click in the virtual phone area, to spot users that quickly navigate through CHIMP to just redeem their payment. Finally, the two rules are combined for users that skipped the control: at least 100 mouse clicks are required not to discard the user.

5.5 Code Coverage

In this section, it is shown how CHIMP's advanced runtime analysis can be used to characterize app behavior using UI interactions. More specifically, this module is used to calculate the amount of an app's code triggered by both human and monkey interaction. While one would

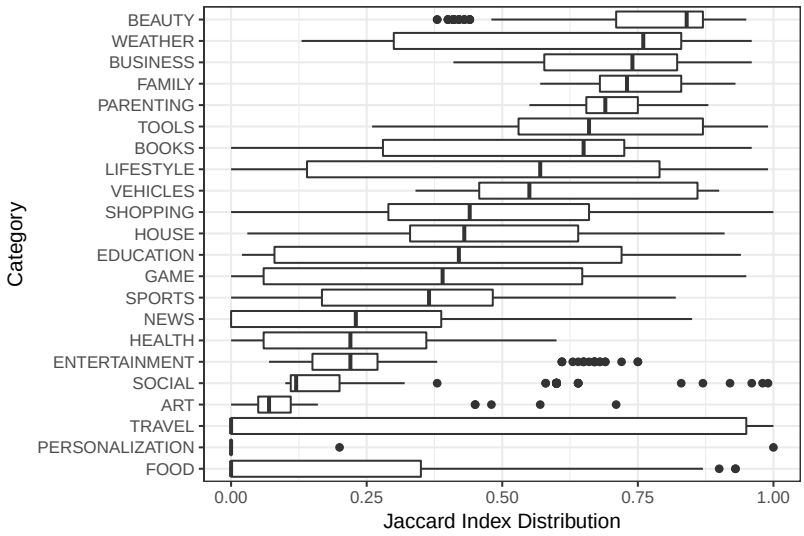


Fig. 5.9 Jaccard similarity indexes between human and monkey runs, per category.

expect humans to perform better for usability tests with specific tasks (e.g., buying a product in a shopping app) and for bypassing known monkey limitations (e.g., login screens, forms, and timing events in interactive apps), one expects monkeys to perform better in exploring overall app code, mostly due to their higher input frequency.

Code coverage is a widely used metric in UI automation literature [16, 18, 39, 100] to compare different exploration approaches. Unfortunately, research literature tends to focus on opensource applications, while CHIMP must also operate with closed-source apps, often with orders of magnitude more complexity. This limitation is mostly due to the ease of analyzing open-source apps' code and the existence of coverage instrumentation tools such as `EMMA` [52]. Standard build tools (which require source code) can be used to run unit tests and calculate coverage using `EMMA`. Unfortunately, to get the coverage of a third party app or to use `EMMA` with the monkey, modifications are required (i.e., adding code to the app). Nevertheless, 59 apps used in Choudhary et al.'s [40] benchmark were analyzed, which in turn were taken from previous literature [16, 18, 39, 100], and tested them with real users (*Code Coverage* in Table 5.1). A similar experiment methodology as [40] was used but with ten, 6 minute ([40] claims no significant coverage improvement after 5 minutes) monkey runs for each app; humans interacted with the app for 84s on average. It was found that humans improved coverage over the monkey for around 40% of apps, while the combination of human and monkey interactions improved coverage for over 60% of apps. Unfortunately, upon closer inspection it was also noticed that these apps were very simple (a median of

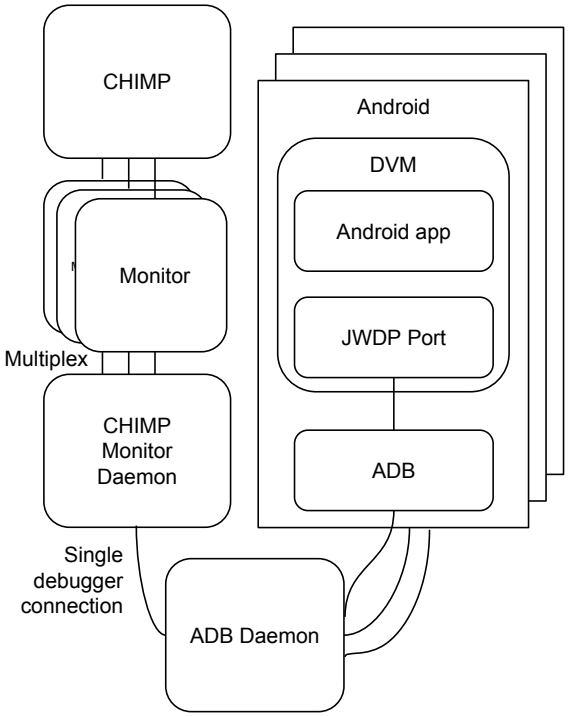


Fig. 5.10 CHIMP's tracing mechanism diagram.

44.5 classes in their main packages) with very limited interactivity (a median of only 4 Activities[12]), and at least 2 apps were completely non-interactive.

To address these limitations, CHIMP uses a different approach (depicted in Figure 5.10). In Android, each app runs on a dedicated Dalvik VM (DVM) that opens a debugger port using Java's Debug Wire Protocol (JWDP) managed by the Dalvik Debug Monitor (DDM). As long as the *device* is set as debuggable (`ro.debuggable` property), one can connect to the VM's JWDP port to activate VM level *method tracing*. Enabling this tracing in Android can be achieved by using either Android's Activity Manager (AM) or a DDM Service. Both end up enabling the same functionality – i.e.,`startMethodTracing` on `dalvik.system.VMDebug` and `android.os.Debug` – but through different approaches. More specifically, AM (via `adb am`) exposes a limited API that eventually reaches the app via Inter-Process Communication (IPC), while the DDM Service (DDMS, as used by Android Studio) opens a connection directly to the VM's debugger, providing fine grain control over tracing parameters. The second approach was chosen since it is parameterizable, thus allowing the definition of the trace buffer size—which by default (8MB) can only hold a few seconds of a trace. Hence, a new DDM Service was implemented, using the ddmlib library [65] to communicate with the VMs and activate tracing.

[12]Activities are responsible for displaying the interactive windows presented to users.

CHIMP's DDM service (CHIMP Monitor Daemon in Figure 5.10) multiplexes all tracing requests through a single debugger, and the ddmlib tracing methods were modified to dump the traces in the VM file system, and set the trace buffer size to 128MB. Since apps can often generate more than 128MB traces, a new background job was added that retrieves and removes traces from the VMs every 30s. Besides preventing the tracing buffer from filling up, this allows the capture of partial traces for apps that might crash during stimulation.

After collecting the apk traces, the output of `dmtracedump`, which generates call-stack diagrams from traces, was parsed to retrieve runtime method invocations of the app. Additionally, the original apk was also decompiled, retrieving its method signatures, including argument and return types (to address method overloading). Matching the class and method signatures of the traces to those extracted from the apk allows one to calculate the class and method coverage of the run. Note that while this method allows calculating the coverage of apps without source code access, it does have shortcomings compared to `EMMA`. First, if the app crashes no trace is generated. Second, it cannot provide code coverage based on lines of code, and thus coverage numbers might not perfectly map to those produced by `EMMA` (e.g., lines of code per method tend to follow a skewed distribution).

To test the tracing mechanisms with popular apps, the top 100 most downloaded apps across Google Play store categories (except widgets, wear, and demo due their low interactivity) were collected. From these, those requiring login or unavailable hardware (e.g., apps using the camera or the device speakers) were filtered, ending up with 55 apps (*Trace Coverage* in Table 5.1). Note that humans could have been given login accounts, but the aim was to perform a fair comparison with the monkeys. These apps have a median of 11,532 classes, 73,958 methods, and 39 activities (with some apps having up to 256 activities); more than one order of magnitude higher than the open-source app set.

Contrary to the initial expectations, humans performed well in regards to code coverage compared to monkeys (Figure 5.8). Humans' median coverage outperformed monkeys for 63.6% of app categories, and by combining both humans and monkeys, coverage can be improved by up to 25%. Another observation is that individual humans and monkeys covered code tends to be quite different, with a similarity lower than 0.5 for 59.1% of the categories (Figure 5.9).

The main take-away is two fold. First, CHIMP can effectively integrate and combine the benefits of different UI interaction tools with competing code exploration techniques. Second, CHIMP's tracing can be useful to benchmark different UI automation techniques with the benefit of being able to test real, unmodified market applications. Furthermore, its tracing capabilities provide a better understanding of app's behavior and its inclusion motivates the use of CHIMP for other purposes, e.g., extending recent malware analysis literature [104] to

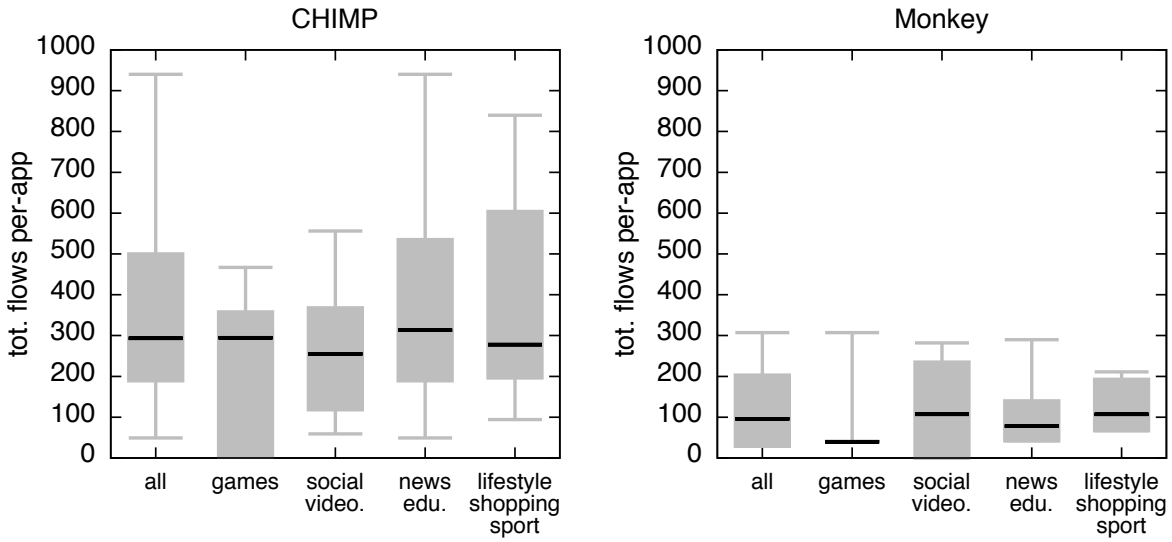


Fig. 5.11 Comparing per-app number of flows in "traffic classification" campaign.

analyze apps at runtime by inspecting traces (instead of performing static analysis only), or even to evaluate malware behavior differences when interacting with humans [13]. Additionally, CHIMP's similarity metrics could also potentially be used to train better UI automation tools that interact more like real users.

5.6 App Classification

In this section, it is showcased how to take advantage of the network data collected by CHIMP (pcap files and ground truth collected by polling `/proc/net`) to create a per-app traffic classifier. Many solutions in the literature propose using HTTP meta-data like hostname and user-agent [36, 144, 157], but such approaches are stymied by the increasing adoption of HTTPS. Instead, inspired by recent work [92, 118, 159], CHIMP is used to create "app signatures", i.e., derive a set of network transport level features (packet sizes, handshake characteristics, hostnames, etc.) that uniquely identify each app. First apps are investigated based on their network traffic (§5.6.1). Then this characterization is used to compare real users with respect to monkeys. Next, the knowledge gained is used to build a traffic classifier and evaluate its effectiveness (§5.6.2).

[13] In fact, this is the focus of the work discussed in Section 3.4.2, where CHIMP was used to test and instrument over 5K malicious and benign apps with over 5K human testers.

Appname	Categ.	Downl. (M)	Train (%)		Prec.	Recall
com.bambuna.podcastaddict	news	5-10	1184	(15.6)	95.0	93.9
com.quvideo.xiaoying	video	100-500	972	(12.8)	82.4	87.2
com.zeptolab.ctr.ads	game	100-500	844	(11.1)	88.5	87.7
it.pinenuts.rassegnastampa	news	1-5	799	(10.5)	78.5	82.9
com.mobilonia.appdater	news	1-5	748	(9.8)	79.3	73.4
com.miniinthebox.android	lifestyle	1-5	690	(9.1)	97.9	95.8
com.Love.Collage.Photo.Frames	lifestyle	5-10	668	(8.8)	62.6	69.3
com.eisterhues_media_2	sports	1-5	648	(8.5)	87.6	81.3
com.topps.kick	sports	1-5	563	(7.4)	96.6	96.8
com.mcdonalds.android	lifestyle	1-5	447	(5.9)	92.5	88.9

Table 5.2 Top-10 apps with respect to number of flows, and prediction accuracy of a Random Forest model.

5.6.1 Traffic Characterization

The used dataset was collected from a campaign with 75 apps, 4 steps per session presented in random order, and 500 users who spent about 100s per app (see "Traffic Classification" in Table 5.1). Pcap files are processed using Tstat [49], an open source passive traffic flow analyzer. Tstat generates per-flow logs, i.e., it internally rebuilds TCP connections based on the exchanged packets, and for each connection reports basic information such as (srcIP, srcPort, dstIP, dstPort) tuples, time of creation, duration, general stats (total bytes/pkts, RTT, TCP handshake duration, TLS handshake duration, etc.), and metadata (hostname for HTTP requests, Server Name Indication - SNI for TLS connections, etc.)

Figure 5.11 (left) shows the distribution of the number of flows per app as boxplot. To make things a bit more readable, apps were put into one of four groups based on their category from the Google Play Store: games (10 apps), social and video players (21), news and education (21), and lifestyle/shopping/sports (24). Note how each group has a median of about 300 flows. That said, while the distribution of three of the groups is heavy tailed, apps in the games group tend to have less traffic.

As expected [69, 131], traffic was found to be predominantly HTTPS. Most app groups have more than 70% of their traffic encrypted, with news and education apps having the least ($\sim$60%). Additionally it was found that SNI is not used in about 28% of the flows. SNI is a TLS extension that lets the client indicate, during the TLS handshake, the server's hostname it needs to talk to, and is commonly used by deep packet inspection solutions. This indicates that *SNI-based traffic classification has limited applicability to mobile traffic*.

Next, have monkeys run the same 75 apps. The monkeys perform 10 runs of 6 min each for each app, i.e., monkeys were intentionally setup to run longer than the aggregated workload done by CHIMP users. Figure 5.11 (right) summarizes the network flow analysis from monkey generated traffic. Despite the favorable set up, monkeys produce only 30% of

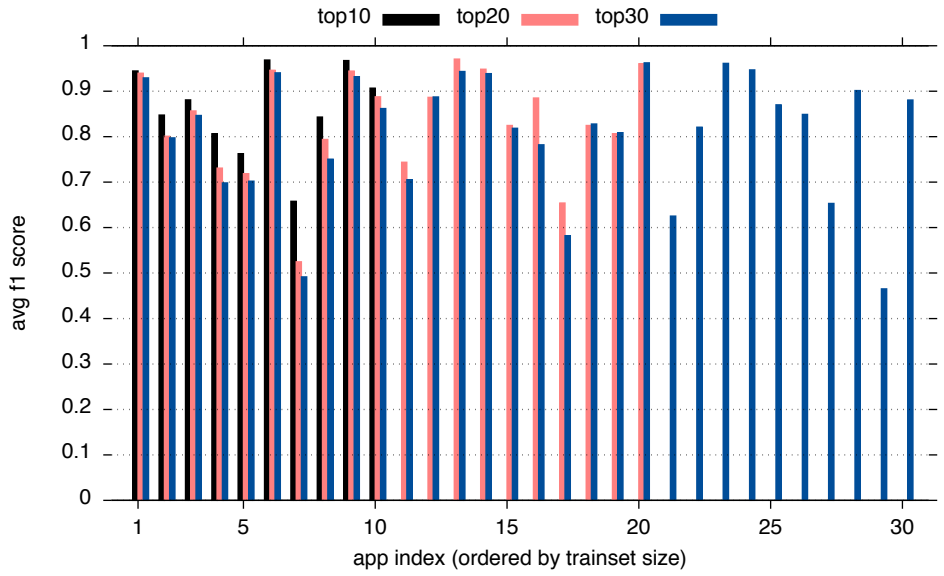


Fig. 5.12 Average per-app f1 score when increasing the number of target classes.

the traffic volume of real users. In particular, note how the games group has the least number of flows. Indeed, the random nature of monkey input is unsuitable for mimicking human behavior for this category of apps.

5.6.2 Random Forest Classifier

Dataset Preparation: The goal is to leverage per-flow and -packet features to train a model for each app. An accurate model cannot be built for apps with little traffic. When ranking apps by their number of flows, the top-10, top-20, and top-30 apps represent 31%, 51%, and 66% of the overall traffic, respectively, and also exhibit more than 400 flows. Thus, in the following experimentation is limited to the top-30 apps since they capture the majority of traffic in the dataset.

The *number of features* for the model is another important parameter for a machine learning approach. In this case, the classic per-flow stats provided by Tstat were used, but with reporting of per-packet level information enabled. Thus one needs to decide how many packets should be used per-flow. It was found that about 80% of flows carry less than 10 packets (detailed analysis omitted due to space limitation) which was adopted as a (reasonable) threshold when extracting features. By coupling per-packet features with those already provided by Tstat, a total of 127 features for each flow are left. Note that this seemingly large number of features is due to the fact that metrics are reported separately for each direction. E.g., the size of the first 10 *outgoing* packets and 10 *incoming* packets are extracted separately.

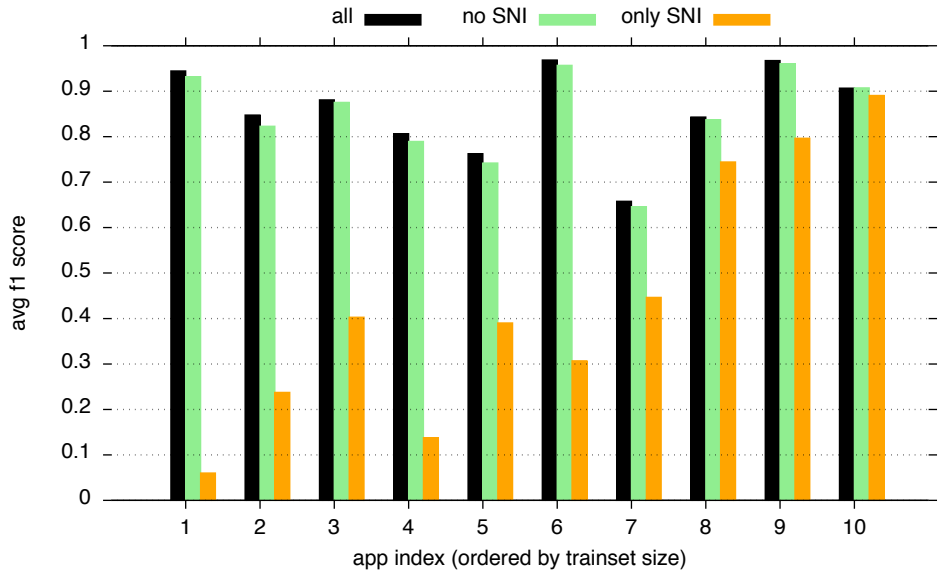


Fig. 5.13 Impact of SNI on classification accuracy.

Finally, each flow in the Tstat logs is mapped to an associated ground truth provided by the polling of `/proc/net`. For this operation the (srcIP, srcPort, dstIP, dstPort, start time) tuple from the two sets of data is matched.

Algorithm Tuning: To model the data, *scikit-learn* was used, a popular machine learning framework. The Random Forest algorithm was used, an ensemble model that combines multiple Decision Trees to mitigate over fitting concerns. This algorithm was configured to use 50 trees (No improvements were observed in prediction quality with larger values).

A 30% split was considered, i.e., 70% of the sample for each app is used for training with the remaining left for testing. The training set was left unbalanced, while the test set was balanced to make use of the standard prediction indexes *Precision*, *Recall*, and *f1 score* (also known as f-measure). Each of these indexes ranges between 0 and 1, and the higher the value, the better the quality of the model.

Finally, for each scenario below, 10 random 30/70 splits were taken and the prediction indexes across the 10 tests were averaged.

Modeling: Table 5.2 lists the top-10 apps in the dataset and the accuracy when classifying them. These apps are very popular (most of them have over 1M downloads), and span several different categories. The model achieves Precision and Recall above 70% for almost all apps.

Further stressing the classifier, Figure 5.12 compares accuracy when classifying 10, 20, and 30 apps, respectively. Only the f1 score is considered in the figure for readability, but results for Precision and Recall are similar (f1 score is indeed an harmonic mean of these two indexes). As expected, the accuracy decreases when increasing the number of apps, but

the drop is minimal. Specifically, on average it moves from 87% when classifying the top-10 to 78% when classifying the top-30. Note that given the unbalanced nature of the dataset, increasing the number of apps reduces the size of the testing set to keep it balanced across apps. Results indicate that this effect does not strongly impact overall accuracy, i.e., the model indeed "fingerprints" each app.

It was found that the SNI is the most important feature in the model, followed by the 1st outgoing packet size. To further detail this, Figure 5.13 compares the f1 score obtained when modeling the top-10 apps with respect to all features, only the SNI, and all features except the SNI. Overall, the SNI alone achieves 51% accuracy across all apps, a very high value (5 times the random base line). However, notice how the importance of the SNI varies across classes. In particular, the classifier struggles with the first few apps in the figure because many of their flows do not have an SNI at all (i.e., the same dummy value is used for the SNI when extracting features for these apps). The combination of transport level features without SNI results in near maximal performance.

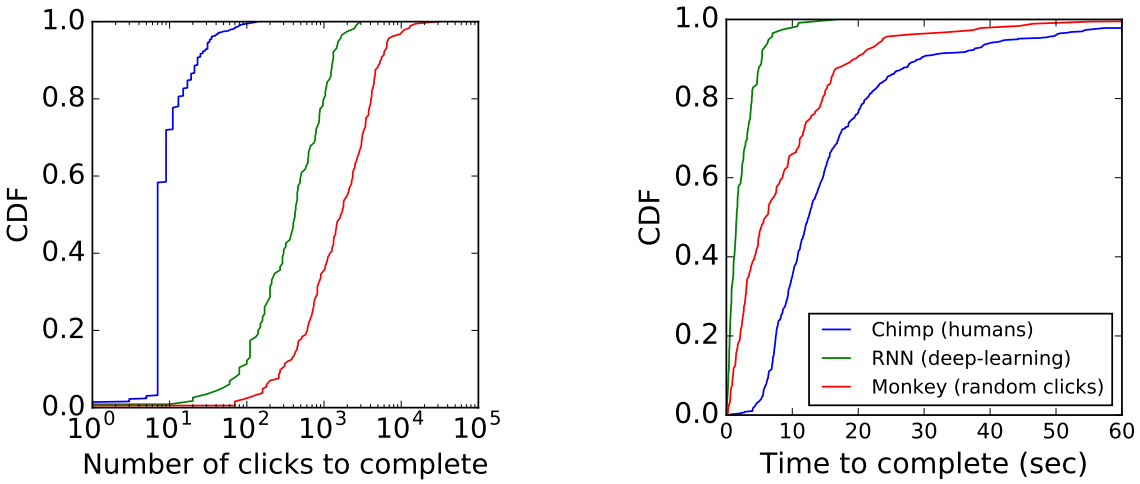
5.7 Discussion & Future Work

Considering CHIMP's input (e.g., multi-touch gestures) and hardware limitations (e.g., sensors), an interesting research direction would be to perform a comprehensive study of how user interactions differ when using real mobile devices. Furthermore, the fact that humans provide less variety in inputs than monkeys means that certain uncommon yet "realistic" combinations of inputs might be missed.

Nevertheless, CHIMP can help improve monkeys by injecting a bit of human intelligence. For instance, a Recurrent Neural Network (RNN [103]) can be trained per app to generate a sequence of mouse/keyboard actions when given user input from CHIMP (x-y coordinates, type of click, timings, etc). Furthermore, app screenshots can be fed into a Convolutional Neural Network (CNN [95]) to identify sequences of visual queues that people typically interact with (e.g., buttons and game elements). Finally, a randomization element can ensure that the resulting model not only mimics humans, but learns from their collective behavior to make educated random guesses about the inputs to generate.

As a proof of concept, the input from 1,000 CHIMP users is used to train an RNN to interact with the control app (§5.4.2). In this app, users had to click three randomly numbered buttons in either ascending or descending order. After clicking the three buttons, they can click either a `<Submit>` button or get a new sequence by clicking a `<Reset>` button.

Both monkeys and the RNN model are unaware of the numerical sequence they have to select. Accordingly, they submit the wrong order 93 and 87% of the time, respectively.



(a) Number of clicks required to complete task

(b) Time it took to complete task

Fig. 5.14 Crowdsourced data can help building smarter monkeys: a recurrent neural network trained with real human inputs significantly outperforms monkeys both in time and number of clicks required to pass a test.

In comparison, humans (who have visual access to the numbers written on the buttons) fail only about 3% of the time (§5.4.2). Monkeys and the RNN model also generate an enormous number of clicks per second to when attempting to pass the control app. Figure 5.14a shows that humans can solve this simple puzzle in less than 7 clicks but monkeys require up to 10,000. Compared with monkeys, the RNN model requires an order of magnitude less clicks. This is because it correctly learned from humans that it has to click on each of the three buttons and then click the submit button. Note that the addition of screenshots would likely further close the gap between the RNN model and humans. Having said that, Figure 5.14b shows that even a simple RNN model speeds up testing time by a factor of 4 when compared to monkeys (1.5 versus 6 seconds). In comparison, humans do take their time to think about their answer (median is 12.4 seconds).

This demonstrates how human input, difficult to collect at scale prior to CHIMP, might lead to the creation of automated testing tools that make more informed decisions about their input.

5.8 Conclusion

In this chapter CHIMP was presented, a system that aims to enable large scale, *human* testing of mobile apps. Its virtual phone environment via which users can interact with Android

from their browser was described in detail, as well as the experimentation platform and data collection modules that make CHIMP a complete system for large-scale app testing with real humans and UI automation tools. CHIMP achieves its scale in part by integrating with paid crowdworker services like CrowdFlower. After evaluating CHIMP, a system calibration was performed, that resulted in guidelines for designing and executing CHIMP experiments.

Next, CHIMP's advanced runtime tracing mechanisms was used to compare humans to the "monkey" UI automation tool, both in terms of code coverage and similarity. It was found that CHIMP successfully leverages the wisdom of the crowd. Its users outperformed the monkey for over 60% of the tested app categories, while CHIMP's combined coverage (monkey and human) improved for the majority of apps, with coverage increasing by up to 25%. Finally, CHIMP was used to capture network traffic generated by apps with the goal of building a traffic classifier. It was found that while monkey inputs were insufficient for generating usable traffic (up to 3 times less traffic volume), a random forest classifier built using CHIMP generated data could reach f1 scores of above 0.9. Overall, CHIMP shows both the *feasibility* and *applicability* of keeping humans in the app testing loop.

5.8.1 Results Summary

Some of the findings of the Chapter 5 include:

- CHIMP was used to compare humans to the defacto standard "monkey" UI automation tool and show that humans generate up to three times more network traffic and outperform the monkey for over 60% of the tested app categories.
- The combined coverage (monkey and human) improves for the majority of apps, with coverage increasing by up to 25%.
- The built random forest network classifier based on CHIMP's testers generated traffic reaches f1 scores of above 0.9.
- Using a prototype RRN that learns from real users, testing tasks can be achieved by up to 4 times faster, while reducing the number of required clicks by an order of magnitude.

Chapter 6

Dynamic Code Execution and Offloading

Motivated by the huge disparity between the limited battery capacity of user devices and the ever-growing energy demands of modern mobile apps, in this Chapter, a new energy saving solution is proposed – INFv. It is the first offloading system able to cache, migrate and dynamically execute on demand functionality from mobile devices in ISP networks. It aims to bridge this gap by extending the promising NFV paradigm to mobile applications in order to exploit in-network resources.

The overall design, state-of-the-art technologies adopted, and various engineering details in the INFv system are presented. A careful study of its deployment configurations is presented via an investigation of over 20K Google Play apps, as well as thorough evaluations with realistic settings. In addition to a significant improvement in battery life (up to 6.9x energy reduction) and execution time (up to 4x faster), INFv has two distinct advantages over previous systems: 1) a non-intrusive offloading mechanism transparent to existing apps; 2) an inherent framework support to effectively balance computation load and exploit the proximity of in-network resources. Both advantages together enable a scalable and incremental deployment of computation offloading framework in practical ISPs' networks.

6.1 Introduction

Pervasive mobile clients have given birth to complex mobile apps, many of which require a significant amount of computational power on users' devices. Unfortunately, given current battery technology, these demanding apps impose a huge burden on energy constrained devices. While power hogging apps are responsible for 41% degradation of battery life on average [116], even popular ones such as social networks and instant messaging apps (e.g., Facebook and Skype) can drain a device's battery up to *nine times faster* due only to maintaining an on-line presence [30].

Recent work has proposed various solutions to offload and execute functionality of mobile apps remotely in a cloud, referred to as a mobile-to-cloud paradigm [38, 41, 45, 73, 89, 93, 161]. Their evaluations have shown that the energy consumption of CPU intensive apps, e.g., multimedia processing apps and video games, can be reduced by an order of magnitude [41, 45]. Beside the extended battery life, there are other benefits, such as faster execution time, responsiveness, and enhanced security by dynamic patching [110].

However, prior work suffers from two limitations. First, they overlooked the potential of exploiting ISPs' in-network resources for functionality offloading, simply using the network as a transmission fabric. Quite different from a decade ago, network middle boxes are no longer simple devices which only forward packets, often featuring multi-core general purpose processors [86] far more capable than those of mobile devices. In fact, many ISPs' own network services have been shifting from specialized servers to generic hardware with the adoption of the NFV (Network Function Virtualization) paradigm[1]. This paradigm can be naturally extended from basic network functions (e.g., packet filtering) to the more general functionality of mobile apps, exploiting "last-hop" proximity to effectively reduce latency, network load, and improve availability compared to a centralized mobile-to-cloud deployment. When deployed close to cellular towers (Radio Access Network), offloading latency could be reduced by up to 86.7%, reducing the energy consumption and execution time by up to 21.6% and 24.5%, respectively, when compared to a popular cloud alternative (Section 6.3.4). Furthermore, such a system could potentially be extended to adapt an app's lifecycle to network conditions, further reducing devices' energy consumption (e.g., delay network dependent background execution [12] in the case of congestion) and the volume of signaling offloaded to the Core Network (CN) [120]. Unfortunately, previous systems either failed to address the challenges of deploying and scaling mobile code offloading systems at all, or overlooked the opportunities to effectively exploit in-network resources. Second, these solutions often utilize intrusive offloading techniques which either require custom OS distributions [41, 73], app repackaging [161], or even alternative app stores, not only increasing security risks and deployment costs, but also greatly increasing the barrier to the market adoption.

This chapter presents INFv, the first mobile computation offloading system able to cache, migrate, and dynamically execute mobile app functionality on demand in an ISP network. It uses advanced interception and automatic app partitioning based on functionality clustering, combined with in-network load balancing algorithms to perform transparent, non-intrusive, in-network functionality offloading. INFv aims to bridge the gap between the

[1]Telefonica aimed to shift 30% of their infrastructure to NFV technologies by the end of 2016 [55, 84, 143]. Other providers such as AT&T [28], Vodafone [54], NTT Docomo [50] and China Mobile [11] are following similar steps.

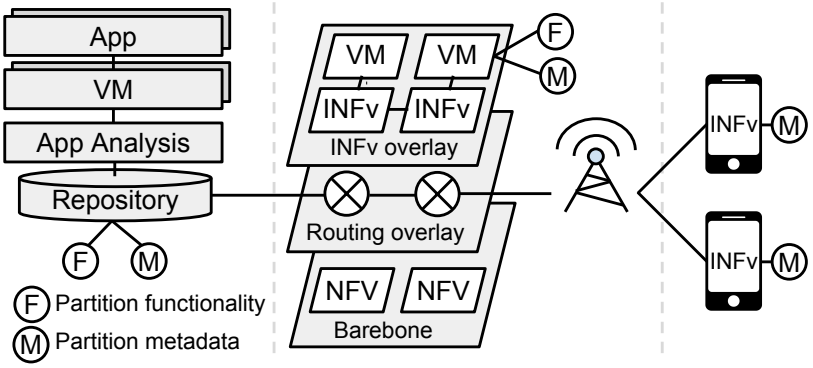


Fig. 6.1 Design of INFv overall system, the three subsystems are divided with gray dashed lines.

limited battery capacity of user devices and the ever-growing energy demands of modern mobile apps[30, 116] by extending the promising NFV paradigm to mobile apps.

While achieving non-intrusive offloading, INFv is able to greatly improve apps energy consumption (reduced by up to 6.9x) and speed up app execution (up to 4x faster). It performs similar to, or better than, local execution 93.2% of the time over 4G while adapting to dynamic network conditions and up to 24.5% faster than a cloud alternative. By providing different strategies to balance functionality load in the network, it reduces both end-user-experienced latency and request drop rates.

Finally, through a real mobile app market study, this chapter shows that the cost of app's storage can be reduced by up to 93.5%, and that top apps have a median of 17 distinct functionality clusters, with up to 57% of offloadable code.

6.2 Design Goals & Architecture

Based on the limitations of previous work, and keeping in mind the recent availability of in-network resources, four major design requirements for INFv were identified, as well as their corresponding challenges.

Instrumenting apps: App instrumentation is needed for profiling and providing offloading capabilities. How can apps be modified without performing app repackaging, using specialized compilers, custom OS or app stores?

Understanding apps: With instrumentation in place, how to detect which functionality is consuming the most energy or executing for longer? Can apps be analyzed without incurring a performance penalty on mobile devices?

Offloading functionality: How to enable apps to make use of remote resources in a transparent and efficient manner with minimal adoption cost? Can offloading adapt to a dynamic network environment and still ensure its energy and performance benefits?

Caching functionality: How to cache functionality in a network, adapting to demand and reducing latency, even for arbitrary network topologies and heterogeneous nodes?

Deployment assumptions: Based on recent/expected NFV/MEC adoption by ISPs, it is the authors belief that these standards will soon be widely available in the ISPs networks. More specifically, and in compliance with the MEC proposal, placed in the Radio Access Network (RAN), where traffic offloading functions can be implemented to filter packets based on their end-point [56, 126] (IP protocol).

The overall INFv architecture that addresses these challenges is depicted in Fig. 6.1. It is divided in three different logical subsystems (separated by dashed lines), each addressing the aforementioned challenges. The leftmost subsystem (Section 6.2.2) profiles and analyzes apps. The rightmost subsystem (Section 6.2.3) provides the on device offloading capabilities. It runs on the user device and executes code on a remote VM in the network. Finally, the third subsystem (Section 6.2.4) runs on the network nodes (NFV) and is responsible for caching functionality and balancing the computation load. Each of the subsystems and their technical challenges are detailed in the next sections.

6.2.1 Dynamic instrumentation

INFv's mechanism to extend app functionality has to have minimal impact and high coverage of devices, i.e., it cannot rely on app repackaging, specialized compilers, custom OS or app stores. To avoid modifying app binaries and change apps' signatures, INFv targets the minimal set of changes required to provide dynamic instrumentation – a reversible binary patch to Android's *app_process*. This process is launched at boot (by the *init.rc* script) and launches Zygote – Android's daemon responsible for launching apps, creating the Dalvik VM and pre-loading Java classes. When a new app is to be launched, this process is forked and the app executes on its own VM with its copy of the systems libraries. By extending this process INFv can add its own classes to the classpath and redirect method invocations to a generic redirection method that allows specifying methods to be intercepted (i.e., hooks).

There are a few dynamic instrumentation frameworks available for Android, that follow similar techniques, such as Xposed [155], Cydia [46] and adbi [8]. INFv is based on the first as it is by far the most popular. As shown in Fig. 6.2, INFv extends it to enable app profiling (Resource Manager – RM), manipulate app lifecycle (App/Thread Manager), perform remote method invocation (Hook Manager and the generic offloading hook – H) and instantiates a custom classloader (*DexClassLoader*) to dynamically load only its signed

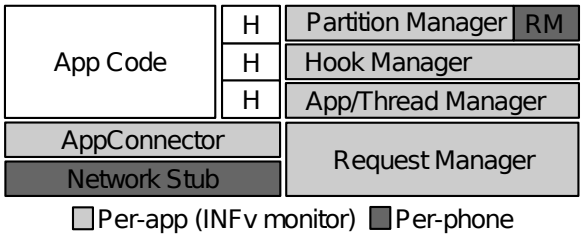


Fig. 6.2 INFv mobile architecture. All components except the Network Stub and the Resource Manager (RM) are contained within the app process.

application strategies and functionality in the backend (Partition Manager). These are detailed in the following sections and their limitations and security concerns discussed in Section 6.6.

6.2.2 Profiling and Partitions

In MCO, apps are typically *partitioned* into a set of functionality to execute locally and another to execute remotely. Such systems need to detect good candidates to offload, e.g., based on energy consumption and execution time. Next section describes how INFv partitions are designed and how the environment-dependent properties (e.g., network conditions) are considered for offloading decisions.

App Analysis & Profiling

Designing functionality partitions requires app knowledge. In INFv most of app analysis is done offline (left-most system in Figure 6.1) and is composed of two steps. The first consists of performing static analysis of the app, i.e., the app analysis platform retrieves an app from the Google Play store, decompiles it (using *apktool*[23]) and parses both the manifest and *smali* files (i.e., disassembled Dalvik bytecode). From the first the app properties are retrieved, such as package name, version and app starting points (e.g., activities, broadcast receivers, etc). From the second a call graph is built where the vertexes represent classes and the edges are method calls. From this call graph one can detect accesses to methods/classes that should execute locally, such as access to local hardware (e.g., sensors) and UI (e.g., Android Views). INFv's static analysis tool (open-source [4, 5]) includes a DSL to tag packages which was used to manually pre-tag most of the Android packages depending on their requirements (e.g., access to hardware).

The second step consists of attributing weights to the call graph edges based on their runtime invocation counts as well as metadata regarding vertices' energy consumptions and execution times. Apps are executed in instrumented VMs and exercised using UI automation (reproducible pseudo-random input [61]). These VMs use the aforementioned instrumenta-

tion mechanisms to load the call graph on app start and intercept method invocations. Some optimizations had to be performed to reduce the tracing overhead, such as increasing the VM heap size and restricting the tracing to app methods. The average of the intercepted methods parameters are registered in the call graph metadata – state size, thread, execution time, invocation counts – along with the predicted energy consumption based on the PowerTutor model [160]. This model uses the device states (CPU, screen, GPS, connectivity) to predict energy consumption. It can derive a per device power model (with a low error of up to 2.5%) by performing regression over the energy discharge patterns observed while looping through different device power states. The INFv cluster can accommodate physical devices to automate this process, or a one time power discharge execution can be run in users devices to retrieve the specific device model. The minimal information needed by INFv's offloading mechanisms requires on average 3MB per app when uncompressed (based on the top 30 market apps), which is reasonable even considering Android's memory constraints (i.e., small heap size – 48MB in Galaxy S2).

While INFv's approach saves the profiling energy on mobile devices as well as the overhead of method tracing (due to devices limited heap size apps can actually become unresponsive), it does have some limitations. Some of the power model metrics (e.g., screen, GPS) are not accurate or observable in a VM. The VM screen is always on and some sensors are emulated. While the second is unimportant as these should execute locally anyways, the first will incur an higher energy consumption prediction which might prevent some functionality from being offloaded. Furthermore, there might also be apps which behavior changes depending on the connectivity type. Finally, while automated input generation might not cover all app code, a significant amount of research exists to improve code coverage [100, 101].

App Partitions

Partitions define which subsets of functionality should be offloaded. In MCO, partitions are generally static and their outcome is a rebuilt app (potentially two disjoint apps) with some functionality re-purposed for offloading. In INFv's client, partitions are just information regarding the offloading subset, i.e., class names and execution properties (e.g., state, execution time, energy consumption, network conditions). These sets are pushed to the device, loaded on app launch and drive app's execution.

Partitions can be devised on method and thread granularity. The latter incurs an extra cost of synchronization (e.g., thread state, virtual state, program counters, registers, stack), and is often restricted to method boundaries [41]. In order to better integrate with the NFV abstraction, this chapter's solution *is* able to provide the granularity of method offloading.

However, since most mobile architectures apps are developed in class-based object-oriented languages, in practice a class offloading granularity is used as methods often invoke methods of the same class and share class state (e.g., class fields). Furthermore, there is a large overlap of classes and packages across apps, which minimizes the impact of storing app functionality in the network (Section 6.4.1).

One of the main concerns in MCO is the balance between offloading computation and its communication overhead due to remote method invocations. To reduce this overhead, the app call graph (Section 6.2.2), i.e., $G = (V, E)$ where V is the set of app classes and E the invocations between classes, is used to detect communities/clusters (sub-graphs of G) of V based on their invocation counts. INFv uses a community detection algorithm – Girvan and Newman (GN) [59] to detect edges that are likely between communities. To do so it recursively detects the edge with highest number of shortest paths between pairs of nodes passing through it (i.e., the edge with highest betweenness) and removes it. By removing edges with high betweenness, the communities get separated and one ends up with distinct functionality clusters. Community detection was firstly proposed by Zhang et al. [161] which used static analysis invocations as edge weights. The problem is that these do not reflect the actual runtime invocation counts between classes, and so the study relies on a weight heuristic based on class semantic similarity, i.e., class names and their textual contents. Unfortunately, the app study in Section 6.4.1 indicates that 82% of the apps perform class name obfuscation and in some cases even the strings within classes can be obfuscated [75]. As INFv executes apps, it registers the method's runtime invocation counts, bypassing such limitations. The used GN algorithm outputs multiple partitions, i.e., it receives the number of clusters as a parameter, which are iterated with a range from one up to the number of (app-specific) classes in the app. Any partitions that include classes previously tagged by static analysis as non offloadable are discarded as offloading candidates. Section 6.2.3 discusses how these predefined partition sets are picked for offloading at runtime.

6.2.3 Offloading Functionality

Once INFv fetches the partition metadata, it needs to provide offloading capabilities and a decision model to ensure that it executes faster and consumes less energy.

The INFv end device system (Fig.6.2) is composed by one or more monitors and one standalone process that provides a network stub and a resource monitoring (RM) system. Each app process transparently loads and executes an INFv monitor on start. The App Manager intercepts all the app entry points declared in the manifest (e.g., broadcast receivers, activities, etc) and binds to the per-device Network Stub. The Partition Manager (PM) loads the partitions and instructs the Hook Manager to hook their entry and exit points, enabling the

interception of their respective members (methods & constructors). Once a target invocation is intercepted, the PM retrieves the current environment-dependent metrics from the RM and decides whether to offload it or not. If so, a message is created and sent by the AppConnector to the Network Stub service (via IPC) that transparently interacts with the closer network node to execute it. Network nodes abstract the network topology by providing a message queue (MQ) between the stub and the execution backend. Within the MQ abstraction, routing is done using the user, device, app and version IDs, along with the fully qualified member name and its arguments. The Thread Manager keeps a per-thread queue of the offloaded requests and suspends the threads until the invocation result is received or there is an intermediate invocation of a member in the same thread.

The INFv backend subsystem runs the same monitor functionality over a light-weight app process. It loads only the required functionality and listens for incoming requests (Request Manager) from the mobile device. On request it creates class instances and executes their respective methods while managing their state. Although mobile apps are expected to be short-lived (imposed by screen off events and CPU sleep mode), state has to be kept while there are references to it. In INFv, remote class instances are replaced locally with "light-weight" instances of the same class (bypassing their constructors [115]), which, on interception, work as proxy objects. These objects are tracked using a custom weak identity hash map that allows the detection of de-referenced or garbage collected objects, which results in a state invalidation message being sent through the INFv network stub. App crashes or force quit also trigger state invalidation. If the crash is INFv's specific, as apps are isolated, the instrumentation can be disabled for the specific app. Finally, when there are no requests or remaining references the VM can be paused after a certain period. More advanced distributed garbage collection mechanisms can be implemented that address some of aforementioned challenges [6].

While INFv's offloading model resembles Java RMI, not only does Android's Java not support RMI by default, but it generally requires a remote interface (e.g., extending *java.rmi.Remote*). Refactoring classes is possible at runtime using bytecode manipulation but is either expensive if done at launch time or significantly increases the app size by up to 40% [161] when stored.

Runtime Decisions

At runtime the PM decides which of the (offline) pre-calculated partitions should be offloaded. This decision is based on the partition's (offline) estimated execution parameters (Section 6.2.2) and the periodic device measurements and state monitored by the RM – network type, bandwidth and latency.

In previous class offloading research [161], a partition would only be valid for offloading if, for each of its classes, all methods perform faster when offloaded. Unfortunately in pratice this rarely holds, even for CPU intensive classes. For example, simple methods such as an overridden *toString()* or *equals()*, will in most situations perform worse than local when offloaded due to the RTT. In INFv, instead of considering methods individually, for each class the method's execution and energy consumption are aggregated based on their observed method invocation frequency (f_i). Therefore, methods that perform better when offloaded can compensate for the least performing methods *if these do not occur too often*. A class c is valid if: $\sum_{m=1}^{M} f_m * t_{m\_local} > \sum_{m=1}^{M} f_m * (RTT + (i_m + o_m)/r + t_{m\_offload})$, where M represents the number of class methods and f_m the method's invocation count normalized by the total number of invocations observed for all the class methods. The size of the method's input and output parameters are represented respectively as i_m and o_m. These are divided by the transmission rate (r) and, along with the RTT, are only counted for methods interfacing between the local and remote partitions (partition boundary). Finally the local (t_{m_local}) and remote ($t_{m\_offload}$) execution times are a function of the CPU frequency difference between the mobile device and the node.

Classes are similarly validated based on the energy consumption of their edges, except that a method's energy consumption is only considered for edges that bridge partitions. For these methods (technically the edges between such classes), only the state transmission costs and the normalized invocation frequency are used when calculating the class's energy consumption.

INFv validates the pre-calculated partitions by increasing the number of partitions (N) until it finds a valid partition. In the worst case scenario N is equal to the number of classes and valid classes are offloaded individually.[2]

Finally, the existing Android diagnosis resources (`dumpsys`) are used to report potential anomalies (e.g., high energy consumption) and processed (offline) by a negative feedback loop to reduce the deviations between estimated and observed values. An advantage of INFv over app repackaging systems is that it is able to dynamically load new partitions and metadata to handle such cases.

6.2.4 Network Subsystem

The network subsystem abstracts the communication between mobile devices and the executing backend. It can be deployed on ISP's RANs where the latency to the User Equipment (UE) is minimal (15-45ms [94]). Offloading requests share a common end-point IP, which

[2] In practice, a hard limit for N must be decided on. However, note that the median number of classes for the most popular apps $\approx$ 5K (Figure 6.10c).

can be intercepted directly at the base station [56, 126], where INFv terminates the traffic and performs further routing. INFv uses a pub-sub MQ system (as proposed by MEC) to store the processed requests while the routing decisions take place and to perform in-network communication.

Because nodes cache functions, there will be multiple copies in the network. The first job of the network subsystem is to route user requests to the closest copy. Functionality execution consumes both CPU and memory as well as other resources (e.g., bandwidth). INFv focuses on the first two since they are usually the most dominant resources. The second job of the network subsystem is to balance the load of executing functions. The goal of load balancing is achieved by strategically dropping or forwarding computation tasks to other nodes to avoid being overloaded. However, instead of distributing load uniformly over all available nodes, a service is better executed as close to a client as possible to minimize latency.

Centralized coordination is not ideal for practical deployment due to three reasons: 1) A central solver needs global knowledge of a network and maintaining such knowledge up-to-date is costly, 2) the optimal strategy needs to be calculated periodically given the dynamic nature of network and traffic, and 3) there is a single point failure. Therefore, two basic heuristic strategies – passive & proactive – are studied and implemented in INFv.

Both strategies have rather straightforward implementations and try to minimize latency. For the proactive one, a simple $M/M/1$-Processor Sharing (i.e., $M/M/1-PS$) queuing model is used to estimate the future queue length. The workload on each node can be further estimated based on the predicated queue length and periodically measured CPU and memory consumption of each function. Next the core idea behind each heuristic is sketched; please refer to [154] for algorithmic details and analysis.

Passive Control: Nodes execute as many requests as possible before being overloaded. If the node is overloaded, the request will be passed to the next hop along the path to a server, or dropped if the current node is already the last hop in the ISP network.

Proactive Control: Nodes execute requests conservatively to avoid being overloaded. To do so, a node estimates the request arrival rate to further estimate the potential consumption. If the estimate shows that the node may be overloaded, it executes some and forwards the rest to the next hop neighbor with the lightest load. NB: This strategy requires exchanging state information within a node's one-hop neighborhood.

6.3 Architecture Evaluation

First a typical deployment of INFv is described and presented thorough evaluations, demonstrating that INFv delivers on its promise of energy savings and faster app execution.

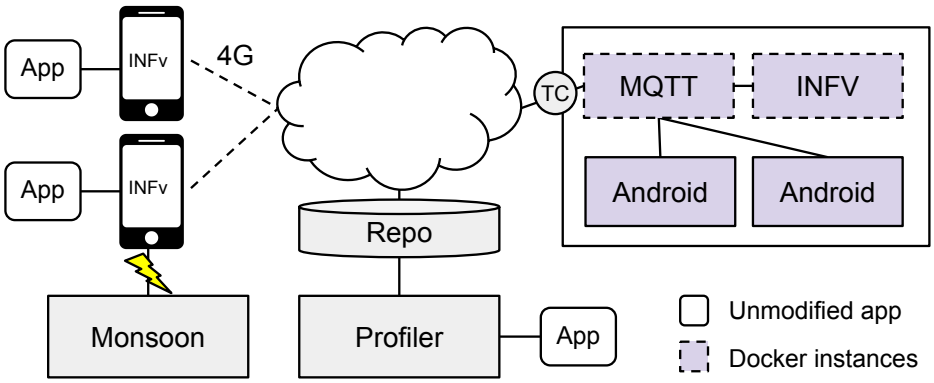


Fig. 6.3 Experimental setup.

INFvs use case: **1)** The profiler analyzes apps and computes partition sets; **2)** The user equipment (UE) installs apps and the local INFv installation downloads their partition metadata; **3)** The UE launches apps, and if the network conditions (bandwidth & latency) are favorable, the execution is offloaded; **4)** the INFv network system forwards offloading request to an available backend, and finally, **5)** the backend executes requests.

The experimental setup is shown in Fig. 6.3. Both INFv's network subsystem and the MQ system are run within docker containers. The selected MQ protocol was MQTT (optimized for mobile devices) and the messages are encoded with protocol buffers [70]. The app profiler utilizes hardware-level virtualization (Android x86) to profile the user apps. Power measurements are taken with a Monsoon Power Monitor and latency is emulated using `TC NetEm` (Traffic Control Network Emulation). In all experiments, phones are factory reset with Android 4.4, no Google account, and INFv pre-installed. In Section 6.3.2 and 6.3.1 a Galaxy S2 (i9100) was used and functionality was offloaded to an Intel Q6600 (4GB of RAM and a 100 Mbit fiber connection). Section 6.3.4 uses a more up-to-date setup: a Galaxy S5 and an Intel i7 4790K (16GB RAM, 300 Mbit fiber). Additionally, while 3/4G setups use real mobile networks, in WiFi the UE and backend share a common WiFi access point.

The offloaded apps were *Linpack* [3], *FaceDetect* [2], *QuickEditor* [67], and *QuickPhoto* [68]. *Linpack* is computationally intensive and *FaceDetect* has high state transmission costs and have been widely used to benchmark MCO performance [93, 135, 161]. *QuickEditor* and *QuickPhoto* both use Google Drive, and although not computationally intensive, they exemplify 1) how INFv can target common functionality across apps and 2) how INFv can provide functionality otherwise absent from a device. Each app is standalone, i.e., no client/server counterpart, and has no special design decisions or implementation to facilitate code offloading.

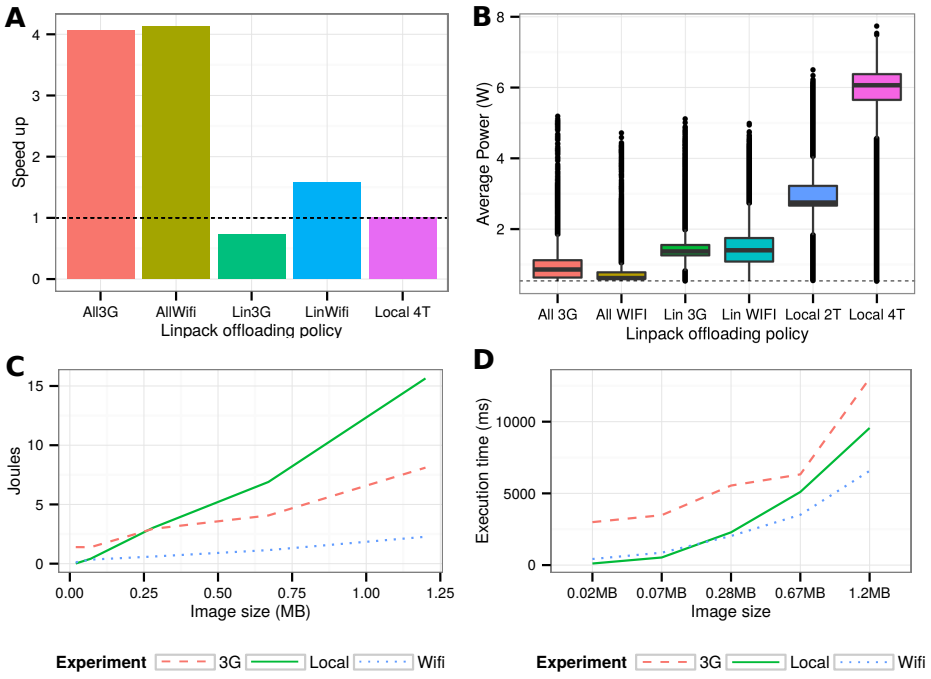


Fig. 6.4 Benefits of mobile code offloading for two apps: Linpack and FaceDetect.

6.3.1 Impact of Code Partitions

Linpack measures a system's floating point computing power by randomly generating a dense matrix of floating point numbers on $[-1, 1]$ and performing LU decomposition for M cycles (iterations). Support for multi-threading was added, a feature many previous automated offloading architectures do not or only partially support [41, 45, 89]. Two different offloading partition sets were tested. For the first (*All*), GN provides two partitions that minimize network communication: 1) a partition that interacts with the UI, therefore invalid for offloading, and 2) a valid partition that performs computation, i.e., all computation is performed remotely and there is almost no communication costs as threading occurs on the backend. The second (*Lin*) represents the worst case scenario by offloading individual classes (i.e. N partitions where N is the number of classes in the app). For *Lin*, only the Linpack class and calculations are offloaded, and thus, threading, as well as pre- and post-cycle processing, occurs on the client with a higher communication penalty. For this experiment (and the next) unconditional offloading (i.e. no runtime decisions) is performed to depict the partition trade-offs.

Fig. 6.4A plots the speed up for the Linpack benchmark running 4 threads, showing that INFv provides the expected computational performance benefits of code offloading. For the *All* partition, INFv achieves a speed up over 4.0x on both WiFi and 3G since there is almost no communication. When the more restrictive partitioning (*Lin*) is used, the WiFi experiment achieves a 1.57x speed up. However, the 3G experiment shows *reduced* performance (0.73

speed up), due to the 3G latency and the high frequency of communication (up to 40 messages, 10 per thread, for each cycle).

Next, Fig. 6.4B plots the distribution of power consumption for both partition sets. The local execution with 2 and 4 threads had a median power consumption of 3 and 6W, respectively; for offloaded executions it was below 2W. For the WiFi *All* partition, the quartiles show a tight distribution of power consumption and, since the UE was mostly idle with a mean energy consumption close to the observed Android background activity (dashed line), the overall energy consumption was reduced by up to 4 times.

Offloading without modularity optimizing can result in reduced performance in high latency networks despite the power consumption reduction. Taken together, Fig. 6.4A and 6.4B, show that INFv's MCO provides real benefits but also demonstrates the trade-offs of different partitionings.

6.3.2 Cost of State Synchronization

The FaceDetect (FD) app, which finds the coordinates of faces in images, is useful for measuring the trade-offs between data transfer and computational speed up. The offloaded partition contains the classes interacting with the face detection APIs and the client device just sends the underlying Android Bitmap object and receives an array of coordinates. The execution time and power were measured from when the app starts up to when the results are drawn on the screen. To test the impact of data transfer, multiple images from the AT&T face database [27] were used, ranging in size from 0.02MB to 1.2MB.

Fig. 6.4D plots the execution time as a function of image size and it can be observed that local execution is faster for images $\leq$ 0.07MB. This is because detecting faces in small images does not have enough computational cost to outweigh the communication costs of offloading. For larger images, the WiFi communication costs are compensated by the VM processing speed. For example, with the 1.2MB image, using WIFI has an execution speed 1.45x faster, but for a 0.2MB image offloading was 3.86x slower. Unfortunately offloading was never justified (in terms of execution time) over 3G due to its high latency.

Fig. 6.4C plots the total Joules FaceDetect consumed, not counting baseline OS consumption. Note that for small images ($\leq$ 0.25MB), local execution results in less power consumption than 3G, although after this point, offloading over 3G saves energy. Offloading over WiFi results in lower power consumption than local execution for all but the smallest image in the dataset. For example, an image with 1.2MB consumes 1.9x less battery when offloaded via 3G connectivity and 6.9x less battery if offloaded via WiFi. In the case of WiFi, the decreased execution time due to the VM processing power is the significant factor in

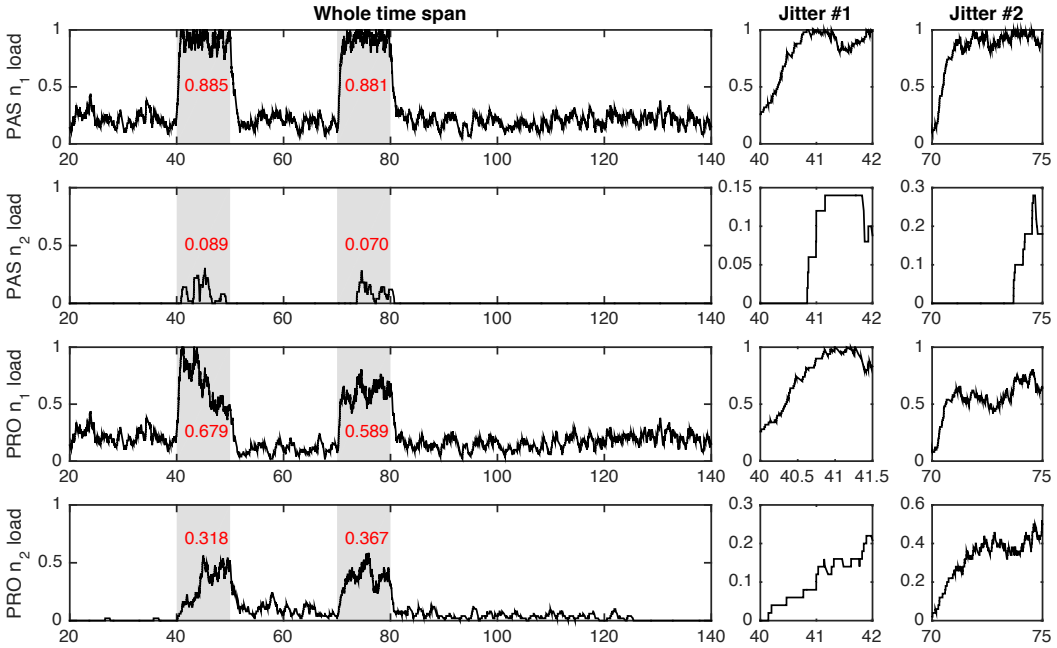


Fig. 6.5 Comparison of two control strategies by examining two adjacent routers: client → router n_1 → router n_2 → server. Two jitter are injected at time 40 ms and 70 ms. x-axis is time (ms) and y-axis is normalized load. Red numbers represent the average load during a jitter period.

energy savings. The 3G energy consumption is higher than WiFi for two reasons: 1) there is additional radio overhead for 3G and 2) the total execution time is larger due the higher RTT.

The main take away from these experiments is that, as in the Linpack experiments, INFv's offloading engine provides both computational and energy benefits. However, if there is substantial interaction between local and offloaded objects that involves passing a lot of data, it can result in a net *loss* of performance. Section 6.3.4 shows how profiling metadata can be used to prevent such scenarios.

6.3.3 Responsiveness to Jitter

Next, we will explore how INFv's control strategies respond to sudden increases in workload (i.e., jitter). To this end, a network topology was set up, composed of a client, two routers (n_1 and n_2), and a server (acting as a catch-all for requests not handled by n_1 or n_2); i.e., client → router n_1 → router n_2 → server. Client's request flow was simulated at a stable rate of $\lambda = 1000/s$ but two instances of jitter were injected at 6λ for 10ms at time 40ms (j_1) and 70ms (j_2). Fig. 6.5 plots the workload over time when the routers use a passive strategy (PASn_1 and PASn_2 in the first two rows) vs. a proactive strategy (PROn_1 and PROn_2 in the

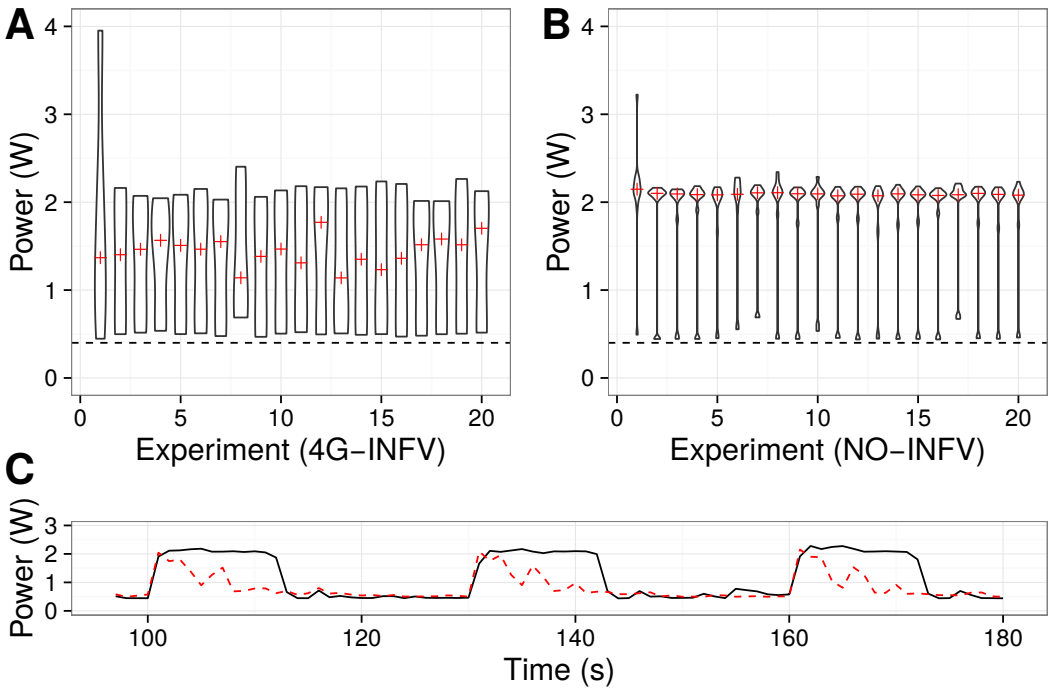


Fig. 6.6 Power consumption distribution (A & B) for 20 executions over 4G, with and without INFv. Crosses represent the median. In C the dashed line represents INFvs' consumed power versus the local execution (continuous).

second two rows). The two right most columns zoom in to the period when j_1 and j_2 have just occurred.

For passive control, PASn_1 takes most of the load (88%), exhibiting consistent behavior for both j_1 and j_2. However, the proactive routers show an interesting variation. For j_1, although PROn_1 successfully offloads 31.8% of load to PROn_2, it also experiences high load for a period of 2ms (row 3, column 2). After j_1, however, PROn_1 enters a conservative mode. Thus, when j_2 arrives, the load curve on PROn_1 is much flatter with no clear peak appearing at all. Instead, it proactively offloads more tasks to PROn_2, resulting in PROn_2 absorbing about 36.7% of the load from j_2. Between 80 and 130ms one can see some load still transferred from PROn_1 to PROn_2 because PROn_1 remains in conservative mode. After 130ms, PROn_1 returns to normal mode and the load on PROn_2 goes to 0.

By checking the second and third columns, we are able to gain an even better understanding on what actually happens when jitter arrives. For both j_1 and j_2, the proactive strategy responds faster; i.e., n_2's load curve rises earlier and faster. For j_2, the proactive strategy responds even faster since PROn_2 is already in conservative mode: PASn_2 only starts taking load at 74 ms, 4 ms later after the j_2 arrives at PASn_1. The major take away here is that INFv is highly responsive to workload jitter due to its network subsystem.

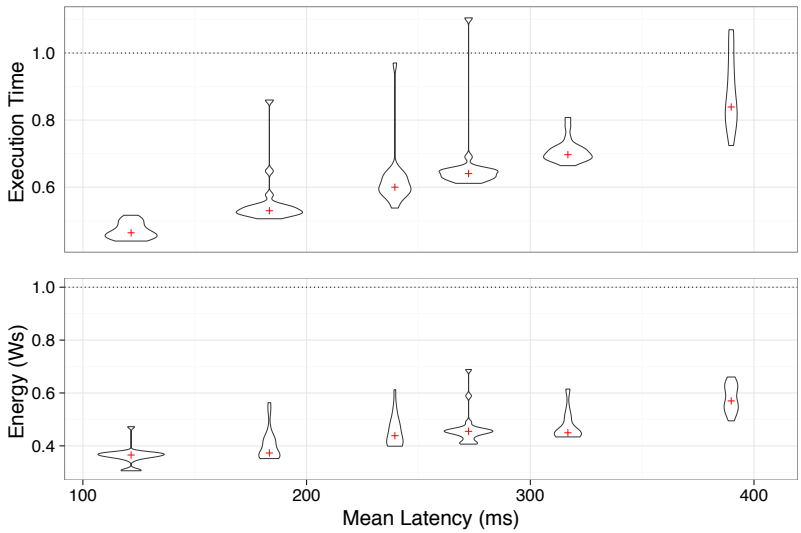


Fig. 6.7 Energy and execution time of FaceDetect execution with INFv enabled. Crosses represent the median. Values are normalized by the mean values of local execution.

6.3.4 Runtime decisions

Finally, INFv's 4G behavior is evaluated, with and without additional induced latency, to stress test INFv's runtime decisions. Fig. 6.6 A) and B) show the observed energy consumption distribution with and without INFv, for 20 experiments using a 1.2MB image [145]. In C) the power over time is depicted for three of these experiments. While the execution without INFv (continuous line), consumed over 2 W for most of the experiment time ($\mu \approx 11.57s$), the execution with INFv (dashed line), is mostly comprised of two spikes in energy consumption. These two spikes represent two distinct phases: 1) image transmission and 2) retrieving and displaying the results; and are dependent on the current connectivity state, e.g., transition to a 4G connected state. In A) and B) it is shown the power distribution for the 20 experiments, with and without INFV, respectively. The power distribution in A) has an higher variance but the majority of the observations are lower than 2W ($\mu \approx 1.35W$ and within a 95% confidence interval of 0.139W) and its execution is up to 2,8 times faster ($\mu \approx 2.3$ times) than the execution without INFv, which results in over 66% energy savings over the 20 executions (idle time excluded). While UEs are becoming more powerful, so are commodity processors and mobile networks, demonstrated by the execution speed improvements in this experiment compared to the 3G experiments (Section 6.3.2).

Fig. 6.7 shows the impact of latency on execution time and energy consumption of FD over 120 executions. The observed latency consists of the latency induced on the backend

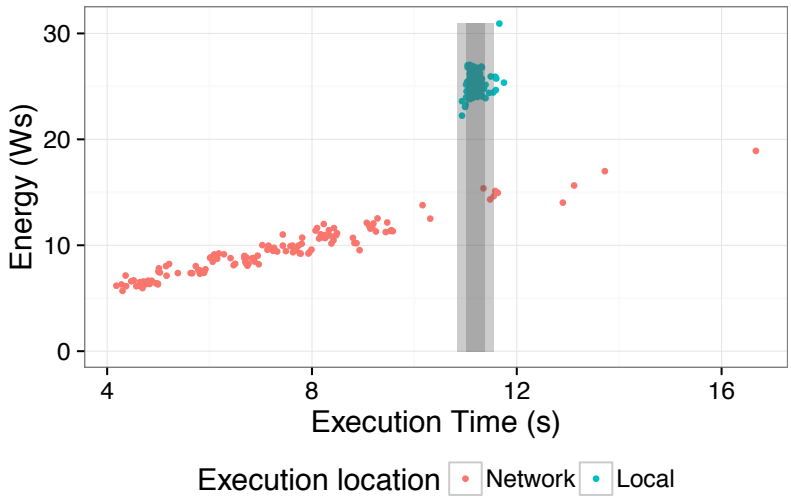


Fig. 6.8 Energy consumption vs. execution time of FaceDetect using INFv under varying latency.

network interface (TC in Fig. 6.3) and the real 4G latency.[3] The energy consumption is always lower when offloading, one can see a reduction in the savings from close to 70% less energy (no induced latency) to 40% due to higher latency. A major takeaway here has to do with in-network vs. cloud deployment: the overall LTE round-trip time (RTT) for offloading to a cloud instance is often over 100ms[4]; quite high compared to hosting the functionality at the ISP's Radio Access Network (RAN) where devices see only 15-45ms RTT [94][5]. I.e., deploying to the RAN can reduce latency by 58.9% to 86.7% (over 90ms difference). Since the results indicate that a 70ms variation in latency can incur up to 24.5% and 21.6% increase in the average execution time and energy consumption, respectively, hosting functionality in-network brings clear benefits. Further, reducing latency via in-network deployment also increases the set of viable candidate apps for offloading to include those that are particularly latency sensitive (e.g., games).

Finally, Fig. 6.8 plots the execution time versus consumed energy for the FD app with the INFv offloading decision model. The RM keeps a rolling window of observed latency (last 3) to the server, which is updated whenever there is no offloading communication (i.e., no threads paused or pending messages) for $> 30s$ and the screen is on. There were 260 experiments over a $>$ 7 hour period (100ms periods). The latency is varied (from 0–600ms,

[3] Note that while 300ms might be unusual in 4G, it is quite common in 3G.

[4] A mean latency of 109.4 and 112.2ms was measured from a mobile device with LTE in Barcelona, Spain to Amazon EC2 regions with the lowest latencies (Frankfurt, *eu-central-1* and Ireland, *eu-west-1*).

[5] The lower bound LTE latency values were confirmed using an USRP transceiver [57] and a conservative software-based LTE protocol stack [114].

with 50ms steps) 30s after each experiment starts. INFv decides whether to offload the face detection partition based on the connection properties at runtime (latency and bandwidth vs. transmitted state) and the profiling estimates (i.e., energy and execution time). Therefore, when executed locally, its behavior should resemble the experiments in Fig. 6.6A.

Note that the majority (86.7%) of local execution times fell within one σ (the dark gray area in the graph) from those of the experiments in Fig. 6.6A, and over 96% of the observed values within two σ (light gray). Moreover, 99% of local executions' energy consumptions were within one σ. There were 4 executions that took *longer* than the local experiments, however, they are an artifact of the periodicity of the latency measurements: INFv was not able to detect the increase in latency prior to making an offloading decision. This is important because such cases can occur due to changes in connectivity (e.g., 4G to 3G) or ISP service degradation. While such impact can be reduced by increasing measurement periodicity, the first is already detected by monitoring changes in the default network interface.

Ultimately, it is clear that, even in the presence of high latency variance, INFv detected when computation should not be offloaded and energy consumption was greatly reduced for all offloaded experiments, performing faster than the worst local execution 98.5% of the time and faster than all local executions 93.2% of the time.

6.3.5 Offloading Popular Libraries and Apps

To evaluate INFv's support for the most popular libraries and apps, first, an ubiquitous library was offloaded, and then, a partitioning analysis of the top free apps in the GP market.

First, the top 2.5K apps were studied and it was found that 76.4% of apps use Google Mobile Services (GMS). Devices without GMS are shutout on a big number of the popular apps on the Google Play Store and so, being able to support them via offloading is a significant endeavour. Two applications were chosen that use a common GMS service – Google Drive (GD). The first, QuickEditor [67], is a text editor that allows users to create, open, and edit text files stored in their GD. The second, QuickPhoto [68], uses the device's camera to take pictures and upload them to GD. To support both apps in a device without GMS installed, 26 GMS classes were offloaded, none of them app specific (code available at [1]). With the in-network solution the number of extra network hops to provide GMS functionality are minimal since GD calls already trigger communication which is forwarded through the RAN. The network communication overhead is also quite minimal: around 46, 50, and 10B, respectively, to create a class, invoke a method, and receive a response.

Second, to address the concern of how many apps are actually offloadable the top 24 GP apps were inspected regarding INFv's partitioning and validation mechanisms (Section 6.2.2). The first finding was that only 6% of app classes extend UI or hardware related classes. While

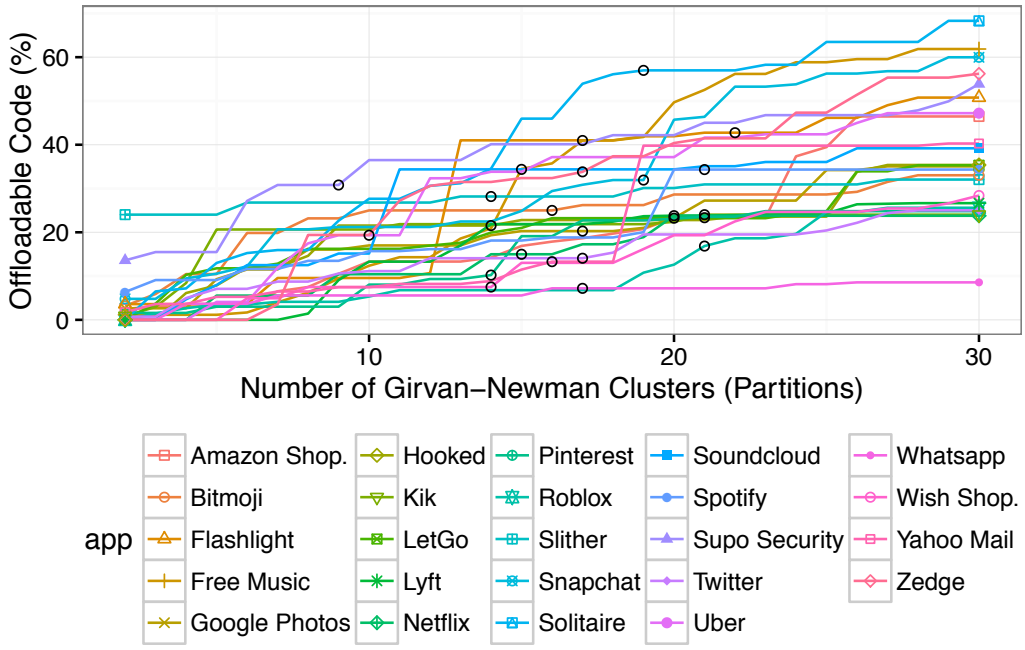
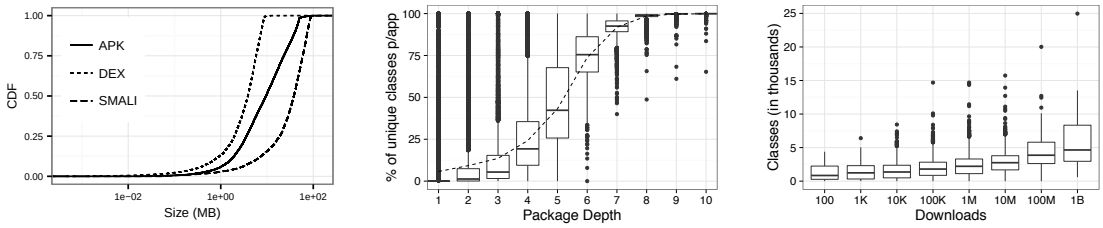


Fig. 6.9 Percentage of offloadable code per number of Girvan-Newman partitions. Black circles represent the optimal partition given by the louvain algorithm.

all other classes could potentially be offloaded, INFv aims to minimize the communication between local and offloaded code. To this end, INFv's dynamic analysis platform was used to execute the apps for 5 minutes each and extract their runtime call graphs to discover valid offloading partitions (i.e., GN communities). Previous work [40] has shown this to be a reasonable interval to achieve high coverage. Classes that communicate often are likely to share a purpose (e.g., handle UI interaction) and so, building communities based on their communication should separate distinct functionality. In fact, Figure 6.9 shows how increasing the number of partitions increases the amount of offloadable code due to this separation of purposes. At 30 GN communities, all but a single app have between 24%-68% of their code suitable for offloading (hundreds to thousands of classes). Using the Louvain [35][6] algorithm to pick the optimal partitioning (based on modularity), one can see that the median number of partitions per app is 17 and that their offloadable code ranges from 7 to 57% ($\tilde{x} \approx 24\%$) with only two apps below 10%. While the benefits of offloading such partitions are dependent on the runtime environment (e.g., state, network connectivity,

[6]Note that in some cases Louvain's might not partition the app in the exact same way, thus its optimal number of partitions should be considered as an estimation. I.e., a number of partitions from which INFv start analyzing the offloading performance.



(a) CDF of the size of app's package – apk, dalvik executables – dex, and disassembled dex format – smali.

(b) Percentage of unique classes per app as a function of name depth used for comparison.

(c) Number of classes per app as a function of app popularity.

Fig. 6.10 Study of over 20K Google Play apps regarding their size, structure and uniqueness.

etc.) these results indicate that the offloading strategy can be applied to more popular and complex apps with huge real-world user bases.

6.4 Deployment considerations

To achieve low latency when serving an offloading request, functionality should optimally be already present in nodes and it should perform well under varying load. This raises two questions: what is the storage cost of hosting the most popular apps? How well does the network subsystem perform under high-load while reducing the offloading latency?

6.4.1 Storage requirements

This section discusses the intuition behind the idea of network functionality caching and empirically shows its feasibility via a study of over 20K of the most popular apps on the Google Play Store in February 2016. As Fig. 6.10a shows, the market app packages (*apk*) are quite small ($\mu \approx 15.3MB$) even considering the actual install size ($\mu \approx 23.9MB$). Only a fraction of the apk is actually app functionality – dalvik executable (*dex*) –, which contains the app classes. For medium/big sized apps (78% of the apps), when extracted from the *apk*, the *dex* size is on average 34.1% of the *apk* size ($\tilde{x} \approx 25.3\%$). The dataset accounts for over 81% ($\approx 15,000$ apps [151]) of all Google Play downloads and in total these apps require an aggregated storage of 307 GB (*apk* size), which is a manageable size.

The large overlap in app functionality can be exploited to intelligently cache functionality in the network. To understand why and how this is possible, a brief overview of Android app organization and packaging is necessary. Android apps organize functions into packages which are further identified with a hierarchical naming scheme similar to domain names.

Intuitively, the hierarchical naming can help identifying shared functionality across apps. Unfortunately, naming in Android can be affected by obfuscation, a security mechanism that remaps functionality and package names. It makes it hard, if not impossible, to detect similarities based on simple name comparisons. E.g., a package "`com.apple`" from app A and a package "`com.google`" from app B can both be renamed to "`a.a`". But, since in Android obfuscation (i.e., Proguard) names are attributed alphabetically, it was found that it is possible to detect if a package name is obfuscated or not based simply on name length. It was found that 37.5% of apps have potentially obfuscated packages with name "a" (at any given depth), while 82% of the apps have at least one class named "a". By studying the name distribution with a single character at any given depth, it was found that the majority of obfuscated package names have names between "`a`" and "`p`" (8% of all packages). Thus, *all package names including names with just one character, at any given level, were filtered.* Unfortunately obfuscated class names do not follow a similar distribution and such filter would exclude an high number of non-obfuscated classes. The remaining package names were used to estimate functionality similarity. If the same package name exists in two different apps it was considered that the functionality within these packages is similar. Obviously, looking at low depth names ($N < 3$), such as the first depth packages (e.g., "`a`" in "`a.b.c`") which contain all other packages and functionality, many apps are likely to share the same name and therefore, most of the functionality will be considered similar (false positives). Looking deeper into the package hierarchy, however, can greatly reduce the rate of false positives. Fig. 6.10b shows the percentage of unique classes per app based on a comparison on their first N package names. The number of unique classes are calculated as the *total number of classes* in the app minus the number of classes belonging to *non-obfuscated* package names that also exist in at least one other app. Considering that the package name distribution has $\mu \approx 4.7$, $\tilde{x} \approx 5.0$ and $\sigma \approx 1.6$, and that most package names have a depth between 4 (Q1) and 6 (Q3), even doing a conservative comparison of packages based on their first 5 name depths (50th percentile), one can see that *only 47% (mean) of apps' classes are unique*. Note that while higher values ($N \geq 8$) reduce false positives, they also increase the number of false negatives as classes within packages with smaller depths are considered as unique. For a depth of 4, which is likely to include the name of the app and respective developer (e.g., "`com.facebook.katana.app`"), 75% (median) of apps' functionality is common with at least one other app. The analysis thus indicates that there is a substantial app's functionality overlap in the Android ecosystem.

Extracting the app classes (*classes.dex*), the storage requirements to host over 81% of the most downloaded apps, are already reduced by 74% ($\approx$ 80GB). If common functionality is co-located, the total reduction can be up to 93.5% ($<$ 20GB, based on the 4th depth median

overlap). While the app analysis platform requires at least the full *apk* files for analysis, the class overlap provides a unique opportunity for deployment in a modern ISP network, allowing INFv to exploit the network topology and ensure that functionality is available as close to users as possible.

6.4.2 Scalability to Workload

In this section a large network simulation is performed using a realistic ISP topology (Exodus [139]) with Icarus [130]. A Poisson request stream is used with $\lambda = 1000/s$ for the arrival rate; increasing the request rate introduces more load to the network. To simplify the presentation, an assumption is made that CPU is the first bottleneck in the system for computationally intensive apps and all experiments are performed >50 times to ensure the reported results are representative.

Fig. 6.11 shows the results of using three strategies (one for each row) with three workloads (one for each column). There are 375 nodes in the network and 100 nodes of degree one are randomly selected as access points to receive user requests. The average load of each node is normalized by its CPU capacity and only the top 50 heaviest loads are presented in a decreasing order. By examining the first column, one can see all three strategies have identical behaviors when the network is underutilized with a workload of λ. The heaviest loaded node uses only about 60% of its total capacity. However, when the load increases to 4λ and 8λ, the three strategies exhibit quite different behavior. The experiment without a control strategy ("none") at the first row, the figures remain the similar shape. Since no load is distributed and a node simply drops all requests when being overloaded, it leads to over 54% drop rate with load of 8λ.

For passive control (second row), one can see both the heads and tails are fatter than "none" control, indicating that more load is absorbed by the network and distributed on different routers. This can also be verified by checking the average load in the figure: given a load of 8λ, passive control increases the average load of the network from 0.2305 to 0.3202 compared to using "none" control. However, there is still over 36% requests dropped at the last hop router. This can be explained by the well-known small-world effect which makes the network diameter short, so there are only limited resources along a random path.

Among all the experiments, a network with proactive control always absorbs all the load, leading to the highest average load in the network which further indicates the highest utilization rate. As the workload increases from λ to 8λ, average load also increases accordingly with the same factor. One very distinct characteristic that can be easily noticed in the last two figures on the third row is that the load distribution has a very heavy tail. This is attributed to the proactive strategy's capability of offloading to its neighbors. It

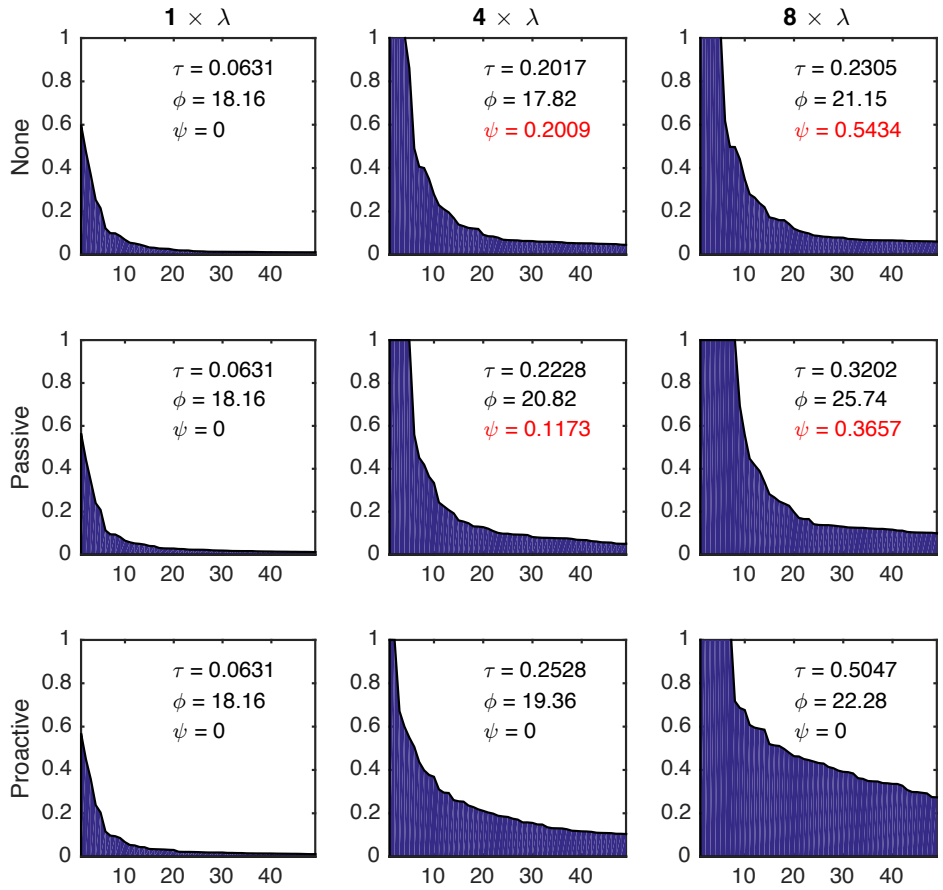


Fig. 6.11 Comparison of three control strategies (rows) on Exodus ISP network, the load is increased step by step in each column. *x*-axis is node index and *y*-axis is load. Top 50 nodes of the heaviest load are sorted in decreasing order and presented. Notations in the figure: τ: average load; ϕ: average latency (in *ms*); ψ: ratio of dropped requests.

is also worth pointing out that only the latency of those successfully executed functions was measured, which further explains why "none" control has the smallest latency, since offloaded functionality gets executed immediately at an edge router connected to a client, but *more than half the requests are simply dropped and not counted at all*. Comparing to the passive strategy, the proactive strategy achieves lower latency. Further investigation on other ISP topologies shows that latency reduction improves with larger networks.

6.5 Limitations

There are some limitations and caveats which deserve further investigation in the future. First, INFv requires attention to the security and privacy of communication and offloaded functionality. While it provides isolation and detects the use of critical OS APIs, in the

future techniques for detecting vulnerabilities/malware [104] and access to privacy sensitive information [53] might be introduced. Although it does not require a custom OS, a one-time root is required, which can be disabled after install. If deployed by an ISP, it can be pre-installed on devices or installed in stores. For other scenarios, either root would be required or, an existing vulnerability could be leveraged to install INFv and secure the device [110].

Additionally, an iOS implementation should be possible using similar interception mechanisms [46] and static analysis can be accomplished by dumping the decrypted apps from memory [141]. There is also the plan of exploring different interception techniques to better support native code [46].

6.6 Conclusion

Battery is a huge constraint for mobile devices and the ever growing demands of computation on limited capacity are unlikely to disappear any time soon. Meanwhile, in-network storage and computation resources are growing. In this chapter INFv was proposed to exploit in-network resources for mobile function offloading. Its data-driven design and implementation were detailed, based on a large scale analysis of a real app market. Our evaluation demonstrates that INFv's non-intrusive offloading technique can significantly improve mobile device's performance (up to 6.9x energy reduction and 4x faster) and effectively execute functionality in the network while reducing latency. Our analysis shows the potential for functionality caching and popular app offloading, while also providing interesting insights into Android apps' obfuscation and composition. INFv is a working system and many of its components are open-sourced [2–5].

6.6.1 Result Summary

Some of the findings of the Chapter 6 include:

- The proposed offloading mechanism is shown to be able to greatly improve apps energy consumption (reduced by up to 6.9x) and speed up app execution (up to 4x faster).
- The runtime decisions perform similar to, or better than, local execution 93.2% of the time over 4G while adapting to dynamic network conditions and up to 24.5% faster than a cloud alternative.
- Through a real mobile app market study, it is shown that app's storage cost can be reduced by up to 93.5%, and that top apps have a median of 17 distinct functionality clusters, with up to 57% of offloadable code.

Chapter 7

Conclusion

This thesis presented a bottom up approach to detect computation and energy inefficiencies via large scale static and dynamic analysis, and mitigating them by transparently offloading functionality in the network.

7.1 Android Execution Scheduling

First, it delved into the challenges in studying **what** apps can do from the perspective of execution scheduling. Research on energy efficiency in mobile devices tends to propose solutions focused on batching activity to amortize the cost of waking up the mobile device and its radio. The efficiency of such solutions depends on the ability of the operating system to schedule background activity at the most appropriate time.

To understand apps' execution scheduling (via alarms), the Google Play store was crawled and over 22 thousand of the most popular apps were downloaded. It was shown that nearly half of apps define their alarms to be *non-deferrable* by the operating system, thus hamstringing Android's ability to optimize scheduling at all. When examining the prevalence of alarms, it was found that they existed across all categories of apps with some apps having up to 70 alarms declared. For apps with alarms, 22.5% have them defined by 3rd party ads/analytics libraries they use, and these libraries account for at least 10.4% of all declared alarms. The inefficiencies of alarms were shown by manually analyzing 60 apps at runtime, finding apps waking up the device an inordinate number of times.

While Android fragmentation has been studied in the past [77, 106, 110], it was generally approached from the perspective of the wide distribution of Android versions, heterogeneous hardware, and lack of updates. In this work a new facet of this problem was revealed: even if the device is supported and up-to-date, apps often target old versions of the Android SDKs, which can have a negative impact on the overall performance of the device. Via static

analysis, it was discovered that a substantial number of apps' alarms are non-deferrable due to targeting older versions of the Android SDK and that by simply changing the target SDK to > 19 these apps would likely benefit from advanced OS alarm scheduling mechanisms. Further, it shows that close to half the alarm API calls are outdated by more than 18 months.

Ads and analytics are a particularly interesting subject of study since they have been shown to have a big impact on energy consumption [76]. In this thesis it is shown that the majority of alarms related to ads and analytics are repeating, meaning that they most likely result in background operations that might have no real end-user benefit. Since a large proportion of Android apps make use of third-party code, future large-scale studies of energy consumption, optimization, and alarm usage should focus on common third-party libraries.

When examining alarm usage at runtime the implications of the static analysis held true for the most part. The apps with the highest number of defined alarms were in fact executing the alarms at an exceedingly high rate. In one egregious case, a single application was responsible for 372 wakeups in a 3 hour period. Overall, the findings indicate that research on energy efficiency on mobile devices needs to incorporate an understanding around the use of alarms.

7.2 Large Scale Dynamic Analysis

After introducing the methods to identify **what** apps can do, this thesis focused on the challenges behind analyzing **how** apps are executing and how to test them with realistic human-like inputs at scale. Existing solutions often rely on distributing specially instrumented apps, or even phones with pre-installed apps, to real users. This approach is quite expensive and has intrinsic scalability limitations.

To solve this challenge CHIMP was presented, a system that enables large scale, *human* testing of mobile apps. A detailed description of its virtual phone environment via which users can interact with Android from their browser was provided, as well as the experimentation platform and data collection modules that make CHIMP a complete system for large-scale app testing with real humans and UI automation tools. CHIMP achieves its scale in part by integrating with paid crowdworker services like CrowdFlower. After evaluating CHIMP, a system calibration was performed that resulted in guidelines for designing and executing CHIMP experiments.

Next, CHIMP's advanced runtime tracing mechanisms were used to compare humans to the "monkey" UI automation tool, both in terms of code coverage and similarity. CHIMP was shown to successfully leverage the wisdom of the crowd. Its users outperformed the monkey for over 60% of the tested app categories, while CHIMP's combined coverage

(monkey and human) improved for the majority of apps, with coverage increasing by up to 25%. Finally, CHIMP was used to capture network traffic generated by apps with the goal of building a traffic classifier. While monkey inputs were insufficient for generating usable traffic (up to 3 times less traffic volume), a random forest classifier built using CHIMP generated data could reach f1 scores of above 0.9. Overall, CHIMP shows both the *feasibility* and *applicability* of keeping humans in the app testing loop.

7.3 Dynamic Code Execution and Offloading

Battery is a huge constraint for mobile devices and the ever growing demands of computation on limited capacity are unlikely to disappear any time soon. Using the combined knowledge of static and dynamic analysis, this thesis focused on the challenges of detecting **when** these executions can be optimized in terms of energy consumption and how to save energy through computation offloading.

A solution was presented for in-network offloading of mobile app computation, along with the key design principles that allow non-intrusive offloading. Unlike previous solutions, this work had a special focus on in-network computation offloading and distribution, proposing and comparing different strategies to effectively balance functionality load while reducing both end-user-experienced latency and request drop rates. Finally, it performed two measurement studies using some of the most popular Android apps. The first studies over 20K apps and shows that there is an high amount of code reuse and that there is potential to greatly reduce the costs of hosting the most popular apps. The second studies a set of 24 apps, showing that community algorithms can be used to partition distinct app functionality and that these apps contain a significant amount of code eligible for offloading.

7.4 Discussion & Future Work

Prior to the study in chapter 4, Google has been giving a lot of attention to background and scheduled execution [63], addressing some of the raised concerns in its latest releases. More specifically, with the introduction of Doze [63], after long periods of inactivity, alarms (including exact alarms) can be deferred. Nonetheless, they also introduced new APIs that still allow the non-deferrable behavior of alarms. While these APIs have more explicit name semantics [1], developers can still misuse them.

One interesting measurement that was not addressed in this thesis is how alarm batching performs regarding mobile network traffic. While the network traffic of batched execution is

[1] e.g., `setExactAndAllowWhileIdle`

likely to occur at similar times, it would be interesting to compare its energy consumption with previous literature that performs network level traffic batching [123].

Regarding the introduction of humans in the testing loops, the study presented in this thesis raises some interesting research questions. First, it would be interesting to study the differences between having users interacting via a browser or a physical device. While most of the touch gestures can be mapped to a mouse (e.g., swipes and clicks), there should be differences in terms of input frequency and precision, as well as other actions that do not map so well to a mouse (e.g., pressure or sensor movements). Characterizing humans' input behavior is interesting for simulating user like interaction, that can be used, for example, in dynamic analysis platforms studying malicious apps that often employ detection evasion techniques (e.g., detecting automation tools [102]).

Another interesting direction briefly presented in this thesis is the use of neural networks for building human-like UI automation tools. In this thesis only human click data (i.e., coordinate, type and time of the clicks) was used to train a RNN [103] to generate sequences of mouse/keyboard actions. Nonetheless, the author believes that using app screenshots to train a CNN [95] to identify sequences of visual queues that people typically interact with (e.g., buttons and game elements), would further close the gap between UI automation and human inputs. Additionally, a randomization element could ensure that the resulting model not only mimics humans, but learns from their collective behavior to make educated random guesses about the inputs to generate.

The traffic classifier in Chapter 5 based on human generated traffic achieved good f1 scores, nonetheless, it would be interesting to test with more apps and test its performance with real traffic (e.g., from a popular proxy or an ISP network).

Finally, with the future developments of MEC and NFV, it would be interesting to perform field tests with the proposed offloading solution and provide a better measurement study regarding the differences between cloud and network offloading. Recent technologies such Anbox [19], and Android's instant apps [62] can make offloading a reality in the future. The first provides a container-based approach to boot Android that does not require an emulation layer and better integrates with the host operating system. The second is Google's recent approach to making Android's apps more modular: app developers break their apps into smaller modules that can be downloaded and installed separately to provide subsets of functionality. It would be interesting to study how the community algorithms in Chapter 6 can be integrated into this technology to automatically generate app modules, and what is the impact of this container technology and smaller functionality sets on the offloading performance.

All together, this thesis presented solutions to analyze and test mobile apps at scale with realistic inputs, using the combined information to offload energy inefficient computation, without a custom OS or app modifications, greatly improving apps' execution time (up to 4x faster) and energy consumption (up to 6.9x energy reduction).

References

[1] 4knahs. Github – gms replacement for google drive. https://github.com/4knahs/gmsreplace.

[2] 4knahs. Github – facedetect. https://bitbucket.org/aknahs/facedetect, 2016.

[3] 4knahs. Github – freelinpack. https://bitbucket.org/aknahs/freelinpack, 2016.

[4] 4knahs. Github – java call graph structure. https://bitbucket.org/aknahs/droidbroker, 2016.

[5] 4knahs. Github – static analysis tool. https://bitbucket.org/aknahs/droidsmali, 2016.

[6] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, Sept. 1998.

[7] AbiResearch. 2q 2014 Smartphone Results. https://www.abiresearch.com/press/2q-2014-smartphone-results-forked-android-aosp-gro.

[8] ADBI. Android dynamic binary instrumentation. https://github.com/crmulliner/adbi.

[9] B. Aggarwal, P. Chitnis, A. Dey, K. Jain, V. Navda, V. N. Padmanabhan, R. Ramjee, A. Schulman, and N. Spring. Stratus: Energy-efficient mobile communication using cloud support. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 477–478, New York, NY, USA, 2010. ACM.

[10] F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan. Energy optimization in android applications through wakelock placement. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 88:1–88:4, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[11] Alcatel. Alcatel-Lucent and China Mobile conduct industry-first live field trial of a virtualized radio access network. https://goo.gl/RcAscW, 2015.

[12] M. Almeida, M. Bilal, J. Blackburn, and K. Papagiannaki. An empirical study of android alarm usage for application scheduling. In T. Karagiannis and X. Dimitropoulos, editors, *Passive and Active Measurement*, pages 373–384, Cham, 2016. Springer International Publishing.

[13] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello, and J. Blackburn. Chimp: Crowdsourcing human inputs for mobile phones. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 45–54, Republic and

Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[14] M. Almeida, A. Finamore, D. Perino, N. Vallina-Rodriguez, and M. Varvello. Dissecting dns stakeholders in mobile networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 28–34, New York, NY, USA, 2017. ACM.

[15] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[16] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[17] Amazon. Live app testing. https://developer.amazon.com/live-app-testing.

[18] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[19] Anbox. Android in a box. https://anbox.io/.

[20] Android-x86. Android-x86 open source project announcement. http://www.android-x86.org/.

[21] Android-x86. Houdini source tree. https://sourceforge.net/p/android-x86/vendor_intel_houdini/ci/kitkat-x86/tree/.

[22] A. Annie. App annie 2015 retrospective - monetization opens new frontiers. http://go.appannie.com/report-app-annie-2015-retrospective.

[23] Apktool. https://code.google.com/p/android-apktool/.

[24] Appetize. Mobile streaming. https://appetize.io/.

[25] Apple. iOS Developer Library – background execution. https://goo.gl/xZd16w.

[26] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese. Radiojockey: Mining program execution to optimize cellular radio usage. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 101–112, New York, NY, USA, 2012. ACM.

[27] AT&T. The database of faces. http://www.cl.cam.ac.uk/research/dtg/attarchive/facesataglance.html.

[28] AT&T. Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf, 2013.

[29] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying online while mobile: The hidden costs. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 315–320, New York, NY, USA, 2013. ACM.

[30] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying online while mobile: The hidden costs. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 315–320, New York, NY, USA, 2013. ACM.

[31] A. Authority. Cyanogen google dependency. http://www.androidauthority.com/cyanogen-google-kirt-mcmaster-582373/.

[32] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[33] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[34] Benedelman. Secret Ties in Google's "Open" Android. http://www.benedelman.org/news/021314-1.html.

[35] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[36] P. Casas, P. Fiadino, and A. Bär. Understanding http traffic and cdn behavior from the eyes of a mobile isp. In M. Faloutsos and A. Kuzmanovic, editors, *Passive and Active Measurement*, pages 268–271, Cham, 2014. Springer International Publishing.

[37] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[38] H. Y. Chen, Y. H. Lin, and C. M. Cheng. Coca: Computation offload to clouds using aop. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 466–473, May 2012.

[39] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[40] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, Nov 2015.

[41] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[42] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2015–2020 white paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[43] Collectd. The system statistics collection daemon. ttps://collectd.org/.

[44] Crowdflower. Crowdsourcing platform. https://www.crowdflower.com/.

[45] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[46] Cydia. Cydia Substrate. http://www.cydiasubstrate.com/.

[47] S. Deng and H. Balakrishnan. Traffic-aware techniques to reduce 3g/lte wireless energy consumption. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 181–192, New York, NY, USA, 2012. ACM.

[48] A. Desnos. Android: Static analysis using similarity distance. In *2012 45th Hawaii International Conference on System Sciences*, pages 5394–5403, Jan 2012.

[49] P. di Torino. Tstat. http://http://tstat.polito.it, 2017.

[50] DOCOMO. DOCOMO Partners with Ericsson, Fujitsu and NEC for NFV Deployment. https://goo.gl/MMQpuC, 2015.

[51] J. D. Drake, Z. Lanier, C. Mulliner, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*.

[52] EMMA. http://emma.sourceforge.net/.

[53] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[54] Ericsson. Ericsson and Vodafone deploy first cloud-based VoLTE. http://www.ericsson.com/news/1993653, 2014.

[55] Ericsson. Telefonica selects Ericsson for global UNICA program. http://www.ericsson.com/news/1988285, 2016.

[56] ETSI. Mobile edge computing. https://goo.gl/rgUi0A, 2015.

[57] Ettus. Ub210. https://www.ettus.com/product/details/UB210-KIT.

[58] Z. Gilani, L. Wang, J. Crowcroft, M. Almeida, and R. Farahbakhsh. Stweeler: A framework for twitter bot analysis. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 37–38, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[59] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[60] Google. Android developer – alarm manager. http://goo.gl/ncrGaO.

[61] Google. Android developer – ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.

[62] Google. Android developers – android instant apps. https://developer.android.com/topic/instant-apps/index.html.

[63] Google. Android developers – optimizing for doze and app standby. https://developer.android.com/training/monitoring-device-state/doze-standby.html.

[64] Google. Android issues – MediaTek Double precision. https://code.google.com/p/android/issues/detail?id=65750.

[65] Google. Ddmlib – apis for talking with dalvik vm. https://mvnrepository.com/artifact/com.android.ddmlib/ddmlib.

[66] Google. Firebase test lab for android. https://firebase.google.com/docs/test-lab/.

[67] Google. Github – quickeditor. https://github.com/googledrive/android-quickeditor.

[68] Google. Github – quickphoto. https://github.com/googledrive/android-quickstart.

[69] Google. HTTPS at Google. https://www.google.com/transparencyreport/https/?hl=en.

[70] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2016.

[71] Google. Arc welder. https://developer.chrome.com/apps/getstarted_arc, 2017.

[72] Google. Arc welder official compatibility list. goo.gl/Q0fy3m, 2017.

[73] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[74] Grafana. The open platform for analytics and monitoring. https://grafana.com/.

[75] Guardsquare. Dexguard. https://www.guardsquare.com/dexguard.

[76] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 100–110, Piscataway, NJ, USA, 2015. IEEE Press.

[77] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*, pages 83–92, Oct 2012.

[78] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[79] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 73–84, New York, NY, USA, 2010. ACM.

[80] T. Hoßfeld, C. Keimel, M. Hirth, B. Gardlo, J. Habigt, K. Diepold, and P. Tran-Gia. Best practices for qoe crowdtesting: Qoe assessment with crowdsourcing. *IEEE Transactions on Multimedia*, 16(2):541–558, Feb 2014.

[81] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Screen-off traffic characterization and optimization in 3g/4g networks. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, pages 357–364, New York, NY, USA, 2012. ACM.

[82] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Radioprophet: Intelligent radio resource deallocation for cellular networks. In M. Faloutsos and A. Kuzmanovic, editors, *Passive and Active Measurement*, pages 1–11, Cham, 2014. Springer International Publishing.

[83] J.-f. Huang. Appacts: Mobile app automated compatibility testing service. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pages 85–90, Washington, DC, USA, 2014. IEEE, IEEE Computer Society.

[84] ICT. Telefonicaś view on virtualized mobile networks. http://www.ict-ijoin.eu/wp-content/uploads/2015/03/6b_Berberana_Telefonica.pdf, 2015.

[85] Intel. Performance results for android emulators with and without intel haxm. https://goo.gl/D6rUf2.

[86] Intel. Hp opennfv and intel onp. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/hp-onp-packet-processing-benchmark-paper.pdf, 2016.

[87] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 253–266, New York, NY, USA, 2013. ACM.

[88] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 1:1–1:5, New York, NY, USA, 2013. ACM.

[89] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: A computation offloading framework for smartphones. In M. Gris and G. Yang, editors, *Mobile Computing, Applications, and Services*, pages 59–79, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[90] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: A case study of android game apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 610–620, New York, NY, USA, 2014. ACM.

[91] K. Kim and H. Cha. Wakescope: Runtime wakelock anomaly management scheme for android platform. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10. IEEE, 2013.

[92] M. Korczyński and A. Duda. Markov chain fingerprinting to classify encrypted traffic. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 781–789, April 2014.

[93] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE INFOCOM*, pages 945–953, March 2012.

[94] M. Laner, P. Svoboda, P. Romirer-Maierhofer, N. Nikaein, F. Ricciato, and M. Rupp. A comparison between one-way delays in operating HSPA and LTE networks. In *2012 10th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*, pages 286–292, May 2012.

[95] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[96] C. Leung, J. Ren, D. Choffnes, and C. Wilson. Should you use the app for that?: Comparing the privacy implications of app- and web-based online services. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, IMC '16, pages 365–372, New York, NY, USA, 2016. ACM.

[97] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[98] H. Liu, Y. Zhang, and Y. Zhou. Tailtheft: Leveraging the wasted time for saving energy in cellular communications. In *Proceedings of the Sixth International Workshop on MobiArch*, MobiArch '11, pages 31–36, New York, NY, USA, 2011. ACM.

[99] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 396–409, New York, NY, USA, 2016. ACM.

[100] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[101] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[102] D. Maier, T. Müller, and M. Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 30–39. IEEE, 2014.

[103] D. P. Mandic and J. Chambers. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. John Wiley & Sons, Inc., 2001.

[104] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *CoRR*, abs/1612.04433, 2016.

[105] McAfee. Mobile Security Report 2014. https://blogs.mcafee.com/consumer/mobile-security-report-2014.

[106] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, Sept 2013.

[107] Metasploit. Google No Longer Provides Patches. https://goo.gl/BcAoRL.

[108] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86, Mar 2014.

[109] Motorola. Android marshmallow moto x 2nd gen. https://forums.motorola.com/posts/86e4e1d737.

[110] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 259–268, New York, NY, USA, 2013. ACM.

[111] NGINX. High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/, 2017.

[112] N. T. Nguyen, Y. Wang, X. Liu, R. Zheng, and Z. Han. A nonparametric bayesian approach for opportunistic data transfer in cellular networks. In X. Wang, R. Zheng, T. Jing, and K. Xing, editors, *Wireless Algorithms, Systems, and Applications*, pages 88–99, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[113] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 389–404, New York, NY, USA, 2015. ACM.

[114] OAI. Open air interface. http://www.openairinterface.org/.

[115] Objenesis. Object instantiation. http://objenesis.org/, 2016.

[116] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[117] L. Onwuzurike and E. De Cristofaro. Danger is my middle name: Experimenting with ssl vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 15:1–15:6, New York, NY, USA, 2015. ACM.

[118] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle. Website fingerprinting at internet scale. In *Proceedings Network & Distributed System Security Symposium (NDSS)*, 2016.

[119] S. Park, D. Kim, and H. Cha. Reducing energy consumption of alarm-induced wake-ups on android smartphones. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 33–38, New York, NY, USA, 2015. ACM.

[120] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal, et al. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[121] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[122] Puma. A Modern, Concurrent Web Server for Ruby. http://puma.io/.

[123] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: Network-wide origin, impact, and optimization. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 51–60, New York, NY, USA, 2012. ACM.

[124] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 137–150, New York, NY, USA, 2010. ACM.

[125] Qualcomm. Developer Network. snapdragon SDK for Android. https://developer.qualcomm.com/software/snapdragon-sdk-android.

[126] RadiSys. Local ip access via home node b. http://go.radisys.com/rs/radisys/images/paper-femto-lipa.pdf, 2011.

[127] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 361–374, New York, NY, USA, 2016. ACM.

[128] T. Richardson and J. Levine. The remote framebuffer protocol. https://tools.ietf.org/html/rfc6143.

[129] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the Android Market. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, pages 113–122, June 2012.

[130] L. Saino, I. Psaras, and G. Pavlou. Icarus: a caching simulator for information centric networking (icn). In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '14, ICST, Brussels, Belgium, Belgium, 2014. ICST.

[131] Sandvine. Spotlight – Encrypted Internet Traffic. https://www.sandvine.com/trends/encryption.html.

[132] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.

[133] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[134] Selenium. Web browser automation. http://www.seleniumhq.org/, 2017.

[135] C. Shi, K. Joshi, R. K. Panta, M. H. Ammar, and E. W. Zegura. Coast: Collaborative application-aware scheduling of last-mile cellular traffic. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 245–258, New York, NY, USA, 2014. ACM.

[136] Sidekiq. Simple, efficient job processing for Ruby. http://sidekiq.org, 2017.

[137] Spice. Spice. https://www.spice-space.org/.

[138] Spice. Xspice. https://www.spice-space.org/xspice.html.

[139] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 133–145, New York, NY, USA, 2002. ACM.

[140] Statista. Number of available apps in the apple app store from jul'08 - jun'16. https://goo.gl/ZmaaRI.

[141] Stefanesser. Dumpdecrypted. https://github.com/stefanesser/dumpdecrypted.

[142] A. Technica. New Android OEM licensing terms leak; open comes with a lot of restrictions. https://goo.gl/iCmRP2, Feb. 2014.

[143] Telecoms. Carta Blanco: NFV at Telefonica. http://telecoms.com/interview/carta-blanco-nfv-in-telefonica/, 2014.

[144] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In M. Roughan and R. Chang, editors, *Passive and Active Measurement*, pages 63–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[145] C. University. The Database of Faces. http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html, 2016.

[146] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. Rilanalyzer: A comprehensive 3g monitor on your phone. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 257–264, New York, NY, USA, 2013. ACM.

[147] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying android applications using java pathfinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[148] M. Varvello, J. Blackburn, D. Naylor, and K. Papagiannaki. Eyeorg: A platform for crowdsourcing web quality of experience measurements. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 399–412, New York, NY, USA, 2016. ACM.

[149] E. J. Vergara and S. Nadjm-Tehrani. Energy-aware cross-layer burst buffering for wireless communication. In *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, e-Energy '12, pages 24:1–24:10, New York, NY, USA, 2012. ACM.

[150] E. J. Vergara, J. Sanjuan, and S. Nadjm-Tehrani. Kernel level energy-efficient 3g background traffic shaper for android smartphones. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 443–449, July 2013.

[151] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.

[152] H. Wang, Y. Guo, Z. Ma, and X. Chen. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 71–82, New York, NY, USA, 2015. ACM.

[153] L. Wang, M. Almeida, J. Blackburn, and J. Crowcroft. C3po: Computation congestion control (proactive). In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ACM-ICN '16, pages 231–236, New York, NY, USA, 2016. ACM.

[154] L. Wang, M. Almeida, J. Blackburn, and J. Crowcroft. C3po: Computation Congestion Control (PrOactive). In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ACM-ICN '16, pages 231–236, New York, NY, USA, 2016. ACM.

[155] Xposed. Xposed framework. http://repo.xposed.info/.

[156] W. Yang, M. R. Prasad, and T. Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In V. Cortellessa and D. Varró, editors, *Fundamental Approaches to Software Engineering*, number 7793 in Lecture Notes in Computer Science, pages 250–265. Springer Berlin Heidelberg, Mar. 2013. DOI: 10.1007/978-3-642-37057-1_19.

[157] Yao, Hongyi and Ranjan, Gyan and Tongaonkar, Alok and Liao, Yong and Mao, Zhuoqing Morley. SAMPLES: Self Adaptive Mining of Persistent LExical Snippets for Classifying Mobile Application Traffic. In *Proceedings ACM MobiCom*, Sept. 2015.

[158] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[159] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[160] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[161] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring Android Java Code for On-demand Computation Offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 233–248, New York, NY, USA, 2012. ACM.

[162] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423, May 2014.