CHIMP: Crowdsourcing Human Inputs for Mobile Phones

Mario Almeida[†], Muhammad Bilal[§], Alessandro Finamore[£], Ilias Leontiadis[£], Yan Grunenberger[£],

Matteo Varvello[‡], Jeremy Blackburn[✡]

[†]Polytechnic University of Catalonia, [§]Universite Catholique de Louvain, [£]Telefonica Research,

[‡]AT&T, [✡]University of Alabama at Birmingham

ABSTRACT

While developing mobile apps is becoming easier, testing and characterizing their behavior is still hard. On the one hand, the de facto testing tool, called "Monkey," scales well due to being based on random inputs, but fails to gather inputs useful in understanding things like user engagement and attention. On the other hand, gathering inputs and data from real users requires distributing instrumented apps, or even phones with pre-installed apps, an expensive and inherently unscaleable task.

To address these limitations we present CHIMP, a system that integrates automated tools and large-scale crowdsourced inputs. CHIMP is different from previous approaches in that it runs apps in a virtualized mobile environment that thousands of users all over the world can access via a standard Web browser. CHIMP is thus able to gather the full range of real-user inputs, detailed run-time traces of apps, and network traffic.

We thus describe CHIMP's design and demonstrate the efficiency of our approach by testing thousands of apps via thousands of crowdsourced users. We calibrate CHIMP with a large-scale campaign to understand how users approach app testing tasks. Finally, we show how CHIMP can be used to improve both traditional app testing tasks, as well as more novel tasks such as building a traffic classifier on encrypted network flows.

ACM Reference Format:
Mario Almeida, Muhammad Bilal, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Matteo Varvello, Jeremy Blackburn. 2018. CHIMP: Crowdsourcing Human Inputs for Mobile Phones. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186035

1 INTRODUCTION

While developing apps has become easier, testing and characterizing them remains challenging because of a dearth of tools for large-scale testing and measurement of mobile apps. The de facto standard app testing technique is to use "monkeys" [12]. 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.

WWW 2018, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/10.1145/3178876.3186035

Figure 1: Human input (100 users) vs monkeys when playing frozen-bubbles.

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 [12], e.g., filling out forms. Second, monkeys' inputs do not reflect those of actual app users. Figure 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 a change in functionality/UI.

For this reason, we still need humans 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) [3, 17]. Indeed, even solutions proposed by app store operators have their own limitations (e.g., in [17] only owned apps can be tested and users must install them). Further, the research literature is littered with examples where authors spend many hours manually running apps to better understand a variety of issues [24, 32, 35].

In this paper we present CHIMP, a flexible Android app testing system that enables quick collection of human inputs for mobile apps. CHIMP runs apps on a server, streaming 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, performance, etc.) as well as explicit user feedback. "Experimenters" (e.g., app developers or researchers) can use CHIMP with apps they want to test and specify the data they want to collect via

campaigns. CHIMP offers integration with CrowdFlower [14], along with user validation techniques, to quickly provide large, trustworthy datasets. For example, the human input behind Figure 1 was obtained from 100 CrowdFlower users in just a couple of hours.

We present the design (§3) and evaluation (§4) of CHIMP with thousands of apps and users. We analyze user interactions via a calibration campaign to design more useful campaigns (§5). We then demonstrate two use cases for CHIMP showing that integrating users into the testing loop can improve code coverage by up to 25% (§6), and that the three times increase in network traffic volume it generates can be used to build an app classification model which achieves f1 scores over 0.9 in some cases. Finally, we discuss our findings, CHIMP's limitations, and conclude (§8).

2 BACKGROUND & RELATED WORK

While CHIMP draws heavily from the automated testing literature, it is designed to meet the needs of researchers in a wide variety of contexts. In this section, we provide 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 App Testing: 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. [12] reviews the state of the art, evaluating 14 testing tools grouped into 3 categories: *random*, *model based*, and *systematic*.

Random tools [5, 27, 38, 50] are best exemplified by the official Android monkey [5]. 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 [2, 9, 11, 20, 48] 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, testing corresponds to exploring the space of the state machine, terminating when all state transitions have been discovered.

Systematic exploration tools [4, 9, 28, 45] are more complicated and use things like evolutionary algorithms in an attempt to produce inputs that improve code coverage. For example, EvoDroid [28] uses the IM as "genes" and the CGM as space to be explored while a fitness function optimizes code coverage and guides the exploration.

All these tools have strengths and weaknesses, but ultimately [12] 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 test" 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 a few services that can bring humans into the app testing loop, similar to CHIMP. The two most popular services are offered by Amazon [3] and Google [17], 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 either require the developer to provide a mailing list or make 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 [8] 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. There are other human-based services occupying the same general space as CHIMP by having in-house app testers with real devices, but scalability and pricing (e.g., [44] charges $99 per user testing session) make it prohibitive for large scale app analysis.

Crowdsourced Systems Although not quite an automated testing tool, Varvello et al. [46] 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 an orthogonal service. Nevertheless, CHIMP's evaluation follows the validation methodology laid out by [46], which includes using engagement estimation, and control questions (§5).

Nikravesh et al. built Mobilizer [31], 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 experiments to gain a view of network conditions across all Mobilyzer devices. They demonstrate ease of development ("about an hour" for a 3rd party app's developers to integrate Mobilyzer) and demonstrate its effectiveness by conducting crowdsourced measurements of mobile Web performance and Video QoE.

While obviously related, Mobilyzer and CHIMP have with fundamentally different objectives. While Mobilyzer provides a previously unseen global view of the mobile network, CHIMP focuses more on app and user behavior, giving researchers and developers a different view of the mobile environment.

CHIMP is designed to complement state of the art tools, and to offer a flexible platform to collect a rich set of data targeting human behavior specifically. 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.

3 CHIMP

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

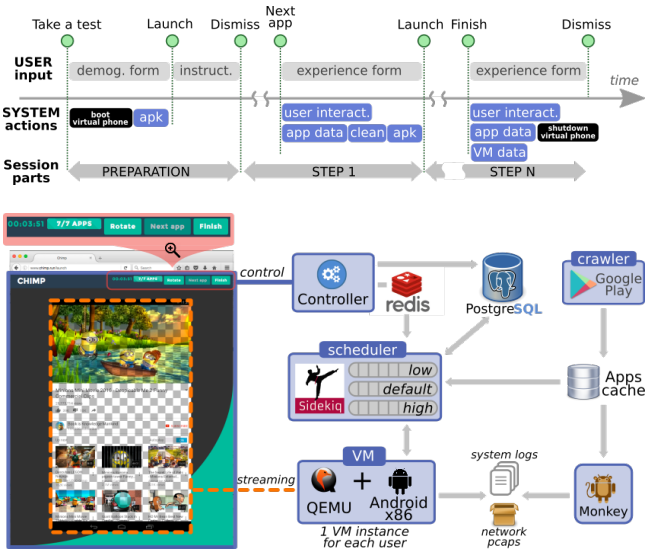

Figure 2: CHIMP**'s architecture: timeline a user session (top), system composition (bottom).**

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 our 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 we discuss 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.

3.1 Web-client Workflow

Figure 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 2 (top) with a timeline, along with the associated system actions (bottom). In the following, we use this visual aid to describe the status of the user's interactions with CHIMP.

We define a *session* 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 (§3.2) 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, we partition each session 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 questionaire* 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. The control area also shows a cumulative session duration and overall step progress (~4 minutes and 7 apps in Figure 2). Multiple user sessions with the same setup are logically grouped into a *campaign* (details discussed in §3.3).

3.2 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. We evaluated several existing solutions and discuss the lessons learned.

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 [22], and is often replaced 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. Since ARM dominates the mobile landscape, most build systems compile to ARM native code by default, leaving 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 our 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 [6]) has native support for houdini [7]. We opted to directly use QEMU, a popular open source hardware emulator and virtualizer. Using QEMU directly gives us 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, particular attention is needed for 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 unusable out of the box. Integrating virglrenderer required us to recompile the Android-x86 image and

QEMU to enable GTK library support. Next, since QEMU emulates a VESA-compliant VGA output device, we modified the VGA BIOS to support WVGA resolutions found in mobile devices (e.g., 400x800).

As reported in Figure 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 [41] and VNC (an implementation of RFB [36]), two popular streaming technologies. Additionally, streaming can also be done via XSpice [42], a variant of Spice which uses an X11 server. We experimented with all three options (results not reported for brevity), by creating an HTML5 client handling the streaming from the virtual phone, i.e., the streaming area in Figure 2. In the end, we opted for VNC which offered a more fluid experience throughout our tests.

3.3 Experimentation Platform

While the AVP is a necessary component for implementing CHIMP, it is not sufficient. CHIMP must allow experimenters 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 (§3.4). 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 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. The controller is a multi-process, multi-threaded, and stateless Ruby on Rails application served from Puma [34] webservers which run behind an Nginx [30] reverse proxy on the server.

We use Sidekiq [40] as our job processing framework, which in turn uses Redis to back job queues. As pictured in Figure 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 reclaiming resources from timed-out sessions and post-processing of campaigns for reporting.

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.

3.4 Data Collection Modules

CHIMP allows the collection of data via different modules. Since CHIMP collects data from human participants[1] we make sure to: 1) collect only the minimum data required for the system to achieve its goals, 2) anonymize any sensitive data, and 3) inform the users of which data is collected prior to each session. Although we intend to expand the type of data collected in the future, for now CHIMP supports collecting six types of data.

User interactions: The web-client collects mouse events from the streaming area, as well as information on browser focus.[2] 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 1), to validate the quality of the crowdsourced workers (§5), or to reproduce crashes by replaying inputs. The web-client uploads user interaction data at the end of each session step (Figure 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). 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, CHIMP retrieves the execution history of the app (`logcat` on the virtual phone), 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 (§6) is captured via method tracing. Additionally, EMMA [16], 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: CHIMP uses *tcpdump* to collect raw pcap files and polls Android's `/proc/net` where open network sockets are listed along with their UID.[3] This solution requires calibration to capture ephemeral flows (i.e., very short lived connections). In our tests, only 1% of the flows were shorter than 100 ms, so we consider a 50 ms polling frequency as a conservative choice. We further map UIDs to the app's package name using the `dumpsys` information.

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, 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` [13] for collection of system metrics and `graphana` [19] for visualization.

[1] All data was collected in compliance with our institutional ethics guidelines.
[2] We do not monitor activity on other browser tabs nor raw keyboard inputs.
[3] NB: in Android each app is given a different numerical user id (UID).

	# 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	76	4

Table 1: Summary of CHIMP's campaigns. NB: CrowdFlower charges an additional 20% processing fee.

4 IMPLEMENTATION

This section describes the details of CHIMP's implementation. We start by briefly reporting on the alternative choices we contemplated. Next, we evaluate our prototype in terms of resource consumption and user experience. Finally, we discuss its limitations.

Setup: We deployed CHIMP 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 we used the default ADB implementation 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, we aim to scale CHIMP horizontally by adding smaller on-demand backend machines and so the following evaluation operates within this limit.

For Android VMs we used a customized Android-x86 image (4.4-RC1, although we also support 6.0), which requires around 1.6 GB after installing when using QEMU's *qcow2* image format [25]. A copy of the image is made for each user; we experimented storing these images both in disk and in volatile memory, i.e., a *ramdisk*. For both setups we benchmarked each of QEMU's caching policies, but in the following we compare only the two best performing strategies: 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, we cast our net wide by crawling the top 500 apps per category in the Google Play store, retrieving a total of 18,787 unique apps. For users, we use both "synthetic" and human users. We created synthetic users with *Selenium* [39], a framework for automating web browser interaction, while we recruited 1K real users on CrowdFlower. We leverage synthetic users to cover the full set of apps (at no cost) while stressing our implementation, i.e., we launch up to 15 simultaneous synthetic users. We leverage real users 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 1 as "Automation" and "Discovery," respectively.

App Compatibility: The first major question we aim to answer is simply how many *real* apps does CHIMP support? Using the app data collection module we find 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 indicates that the app binary is broken, incompatible with hardware, or it takes too long to install (we enforce an upper bound of 40s for app installation). Failing to be brought to the foreground is often a sign of a crashing app. Regardless, 71% of apps were successfully brought to the foreground, indicating that CHIMP has pretty good support for real-world apps (compare to the "official" mechanism [47] for running Android apps in a browser which supports 58.7% of tested apps [26]).

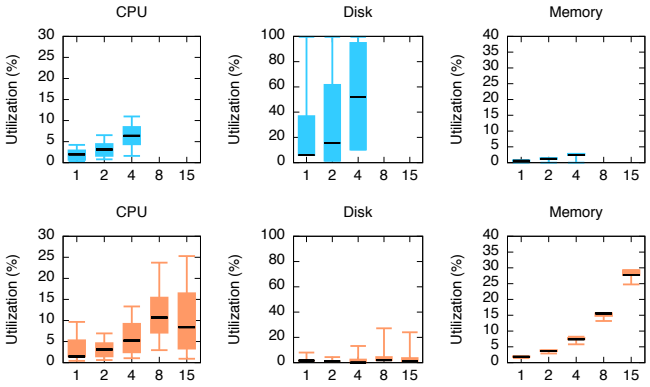

Figure 3: CPU, disk, and RAM utilization as a function of number of concurrent users. Rows are HDD and RAM based disks, respectively.

Resource utilization: Next, to get an idea of how our design choices play out in terms of resource usage, Figure 3 plots the distribution of CPU, disk, and RAM utilization as recorded by CHIMP during our synthetic user campaign. Notice that for disk, the utilization corresponds to throughput of the disk. Our server has plenty of CPU available to sustain the workload, e.g., in the worst case (15 concurrent users), we see a maximum of 30% CPU utilization. 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. tmpfs alleviates this contention and allows CHIMP to scale up to 15 concurrent VMs. Here, 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. We saw a maximum of 10 and 7 Mbps network transmission and reception rates, respectively.

Based on the analysis above, CHIMP uses tmpfs and the controller (§3.3) 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 since it provides no API to request a user arrival rate. However, jobs can be started, paused, and stopped at the job owner's discretion and we engineered the CHIMP controller to regulate system workload by using this "trick."

Service latency: To benchmark CHIMP's performance we collected the boot and app install times 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 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, we see how "fast" app interaction felt to users and if real users were able to trigger any crashes that the synthetic users did not detect. Next, we also investigate 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 4 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

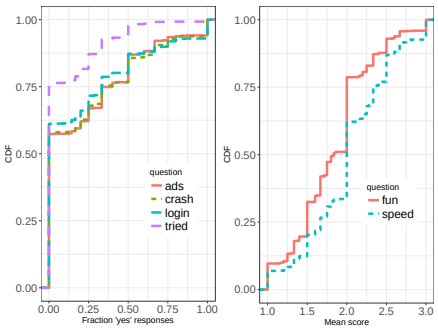

Figure 4: CDF of mean question scores per app for 1,000 tested apps.

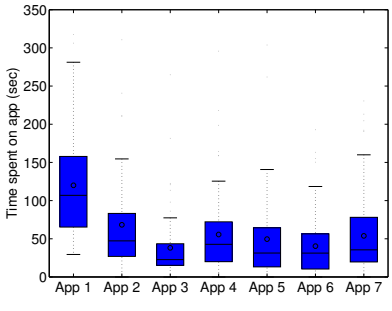

Figure 5: Boxplots of time spent on each app.

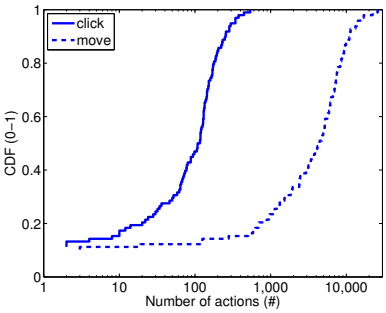

Figure 6: CDF of actions (clicks and move) per user.

on the left while questions scored on a scale of 1 to 3 are on the right. From the left plot, we can see that most of the apps were new to our 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 we see no crashes reported for most of the apps, about 15% of apps have a majority of users claiming crashes. From the plot on the right, we see that users found the apps a bit boring with about 50% receiving a mean score less than 2.0. Regardless, we see that 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, we could not reach a conclusive correlation between system metrics and user's reported app experience. This is a complicated subject since multiple factors need to be considered, e.g.., user connectivity, apps characteristics (e.g., unresponsive apps due to bad design, resource consumption, and/or CDN content policy retrieval), etc. We leave a more careful characterization 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 Windows Mobile. 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 we could have created accounts 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, we expressly instruct users 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 [31]. Additionally, certain experiments may be more longitudinal in nature, while CHIMP's AVPs are bound to a session that will eventually time out.

5 CAMPAIGN CALIBRATION

Apart from the engineering challenge to build CHIMP, a more fundamental challenge is dealing with real people. To collect meaningful data, we need to engineer campaigns people will be happy to take, e.g., select the right number of apps per session. We also need to validate input to avoid random clickers or distracted users.

In this section, we use CHIMP to get insights on how campaigns should be structured. Accordingly, we run a "calibration" campaign (Table 1) with sessions containing 7 steps, i.e., 6 apps plus a "control" one (see §5.2 for details), presented in random order. The apps we test are: 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. We request only users that are "historically trustworthy" 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). We do not enforce a minimum number of steps that users should take, leaving them free to skip steps or even finish without having tested all 7 apps. Similarly, we do not impose any minimum 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, we first investigate user behavior while interacting with CHIMP's website, with the goal of identifying guidelines for future measurement campaigns (§5.1). Then, we compare techniques for discarding unreliable responses (§5.2).

5.1 User Behavior

We start by investigating how users navigate through a session. 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 we reached these (high) utilization numbers through various iterations on the website's design. Specifically, the addition of a progress bar (Figure 2) generated an impressive 40% improvement.

Next, we investigate how much time users spend per app (Figure 5). Note that we subtract any time the user spent interacting with a other browser tabs. 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 our methodology. While regular apps are placed randomly, we opted to show a control app (§5.2) 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 little time on it.

Finally, we quantify how much users interact with apps via mouse movements and clicks. Figure 6 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, we choose to structure our future campaigns with four apps (three plus control). Our rationale is twofold: 1) maximize the number of users who will complete a full session and 2) increase the chance the user will spend meaningful time on an app. Apps are presented in random ordered to make sure they get comparable amount of user time.

5.2 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. We also draw inspiration from the validation methodology in [46] as it dealt with similar issues (§2).

Specifically, CHIMP leverages "engagement" as an indication of user quality. We define engagement by the amount of mouse clicks and movements detected in the virtual phone area. To avoid setting arbitrary thresholds, we discard users who never interact with the phone area, i.e., about 10% of the users (Figure 6). Clearly, users with a single click should also be discarded but we leverage additional filtering techniques to better capture such users.

We also use *control questions*, i.e., questions to which the answer is known a priori [21], to identify low quality users. We developed an Android app 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 a sign of a low quality user. We implemented the control 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 it was well designed; however, 22% of users *skip* the control app. Users failing the control are labeled as low quality and discarded (we still pay these users). Users that skipped it are given a second chance for redemption: they are not discarded if they show high level of engagement with other apps. Specifically, we require such users to have at least 100 clicks across the entire 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 other campaigns in this paper exhibited the same rate of attrition.

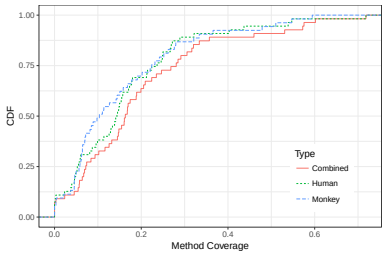

Figure 7: CDF of method coverage achieved by humans, monkeys, and CHIMP's combination of both.

To summarize, we use a "control" Android app of our own design to identify low quality users within a CHIMP's campaign. We also use engagement, 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, we combine the two rules for users that skipped the control: we require at least 100 mouse clicks not to discard the user.

6 APP RUNTIME ANALYSIS

We now show how CHIMP's advanced runtime analysis can be used to characterize app behavior using UI interactions. More specifically, we use this module to calculate the amount of an app's code triggered by both human and monkey interaction. While we 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), we expect 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 [2, 4, 11, 27] 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` [16]. 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 to the app's code are required. Nevertheless, we analyzed 59 apps used in Choudhary et al.'s [12] benchmark, which in turn are taken from previous literature [2, 4, 11, 27], and tested them with real users (*Code Coverage* in Table 1). We used a similar experiment methodology as [12] but with ten, 6 minute ([12] claims no significant coverage improvement after 5 minutes) monkey runs for each app; humans interacted with the app for 84s on average. We 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 we noticed that these apps are very simple (a median of 44.5 classes in their main packages) with very limited interactivity (a median of only 4 Activities[4]), and at least 2 apps were completely non-interactive.

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

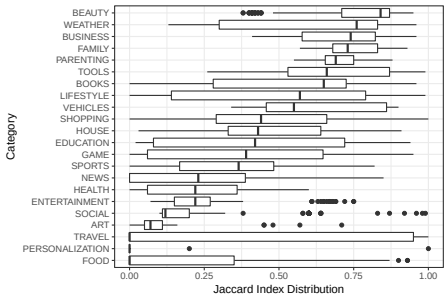

Figure 8: Jaccard similarity indexes between human and monkey runs, per category.

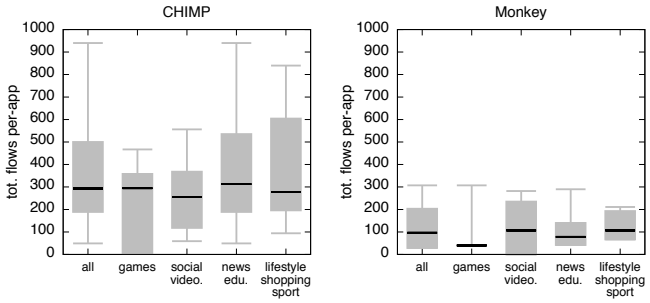

Figure 9: Comparing per-app number of flows in "traffic classification" campaign.

To address these limitations, CHIMP uses a different approach. In Android, each app runs on a dedicated VM 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), it is possible to activate *method tracing* capabilities. To this end, we implemented our own tool to open a connection directly to the VM's debugger (using an Android platform library called ddmlib [1]) and collect traces on a given app. After collecting the app traces, we parse the output of `dmtracedump`, which generates call-stack diagrams from traces, to retrieve the runtime method invocations of the app. Additionally, we also decompile the original app, 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 app binary allows us to calculate the class and method coverage of the run. We note that while this method allows us to calculate coverage of apps without source code access, it does have shortcomings, i.e., 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 our tracing mechanisms with popular apps, we collected the top 100 most downloaded apps across Google Play store categories (except widgets, wear, and demo due to low interactivity). From these, we filtered those requiring login or unavailable hardware (e.g., apps using the camera or the device speakers), ending up with 55 apps (*Trace Coverage* in Table 1). We note that humans could be given login accounts, but we decided to perform a fair comparison with the monkeys. These apps have a median of 11,532 classes, 73,958 methods, and 39 activities (up to 256 activities); more than one order of magnitude higher than the open-source app set.

Contrary to our initial expectations, humans performed well in regards to code coverage compared to monkeys (Figure 7). Humans' median coverage outperformed monkeys for 63.6% of app categories, and by combining both humans and monkeys, we can improve coverage by up to 25%. We also observe that individual humans and monkeys covered code tend to be quite different, with a similarity lower than 0.5 for 59.1% of the categories (Figure 8).

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 us with a better understanding of app behavior and its inclusion motivates the use of CHIMP for other purposes, e.g., extending recent malware detection tools [29] to use runtime analysis instead of performing static analysis only. CHIMP's traces could also be used to train better UI automation tools that interact more like real users.

7 APP CLASSIFICATION

In this section, we showcase 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 [10, 43, 49], but such approaches are stymied by the increasing adoption of HTTPS. Instead, inspired by recent work [23, 33, 51], we use CHIMP 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. We first investigate apps based on their network traffic (§7.1). We then use this characterization to compare real users with respect to monkeys. Next, we use the knowledge gained to build a traffic classifier and evaluate its effectiveness (§7.2).

7.1 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 ("Traffic Classification" in Table 1). Pcap files are processed using Tstat [15], an open source passive traffic flow analyzer. Tstat generates per-flow logs, i.e., it rebuilds TCP connections based on the exchanged packets, and for each connection reports basic information, e.g., (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, TLS Server Name Indication, etc.)

Figure 9 (left) shows the distribution of the number of flows per app as boxplot. To make things a bit more readable, we put apps 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). We first 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 [18, 37], we find traffic to be predominantly HTTPS. Most app groups have over 70% of their traffic encrypted, with news and education apps having the least (~60%). We additionally

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

find that SNI is not used in ~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., we intentionally set up monkeys to run longer than the aggregated workload done by CHIMP users. Figure 9 (right) summarizes the network flow analysis from monkey generated traffic. Despite the favorable set up, monkeys produce only 30% of the traffic volume of real users. In particular, we 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.

7.2 App Classifier

Dataset Preparation: Our 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 over 400 flows. We thus limit experimentation to the top-30 apps since they capture the majority of traffic in our dataset.

The *number of features* for the model is another important parameter. In our case, we use the per-flow stats provided by Tstat, but enabled reporting of per-packet level information. We thus need to decide how many packets should be used per-flow. We find that ~80% of flows carry less than 10 packets (detailed analysis omitted due to space limitation) which we adopt as a threshold when extracting features. By coupling per-packet features with those provided by Tstat, we end up with 127 features for each flow. Note this seemingly large number of features is because metrics are reported separately for each direction. E.g., we extract the size of the first 10 *outgoing* packets and 10 *incoming* packets separately. Finally, each flow in the Tstat logs is mapped to the ground truth provided by the polling of `/proc/net`.

Algorithm Tuning: We use the Random Forest algorithm, which combines multiple Decision Trees to mitigate over fitting. We configured the algorithm to use 50 trees (we do not observe improvements in prediction quality with larger values). We consider a 30% split, i.e., 70% of the samples for each app are used for training and the rest for testing We leave the training set unbalanced, while we balance the test set to make use of the standard prediction metrics *Precision*, *Recall*, and *f1 score* (also known as f-measure). Finally, for each scenario below, we take 10 random 30/70 splits and average the prediction indexes across the 10 tests.

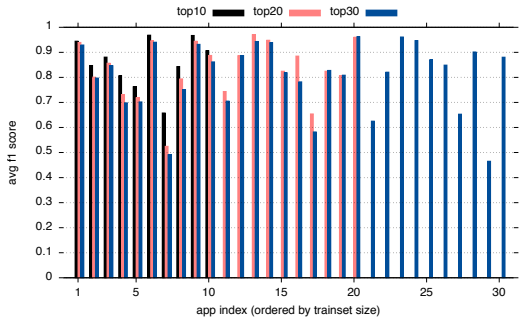

Figure 10: Average per-app f1 score when increasing the number of target classes.

Modeling: Table 2 lists the top-10 apps in our dataset and the accuracy when classifying them. These apps are popular (most of them have over 1M downloads), and span several different categories. The model achieves Precision and Recall above 70% for most apps.

Further stressing the classifier, Figure 10 compares accuracy when classifying 10, 20, and 30 apps, respectively. While we plot only the f1 score for readability, results for Precision and Recall are similar. As expected, the accuracy decreases when increasing the number of apps, but the drop is minimal. On average, it moves from 87% when classifying the top-10 to 78% when classifying the top-30. Given the unbalanced nature of our dataset, increasing the number of apps reduces the size of the testing set to keep it balanced across apps, but results indicate this effect does not strongly impact overall accuracy, i.e., the model indeed "fingerprints" each app.

8 CONCLUSION

In this paper we presented CHIMP, a system that enables large scale, *human* testing of mobile apps. We described in detail the virtual phone environment via which users can interact with Android from their browser, 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. Although CHIMP has some limitations with respect to input (e.g., multi-touch gestures) and hardware (e.g., sensors), an interesting research direction would be to perform a comprehensive study of how user interactions differ when using real mobile devices.

CHIMP achieves its scale in part by integrating with paid crowdworker services like CrowdFlower. After evaluating CHIMP, we performed a system calibration that resulted in guidelines for designing and executing CHIMP experiments. Next, we used CHIMP's advanced runtime tracing mechanisms to compare humans to the "monkey" UI automation tool, both in terms of code coverage and similarity. We have 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, we used CHIMP to capture network traffic generated by apps with the goal of building a traffic classifier. We 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.

REFERENCES

[1] 2017. DDMlib: APIs for talking with Dalvik VM. (2017). https://mvnrepository.com/artifact/com.android.ddmlib/ddmlib.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[3] Amazon. 2016. Live App Testing. (2016). https://developer.amazon.com/live-app-testing.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, 59:1–59:11. https://doi.org/10.1145/2393596.2393666

[5] Android. 2017. UI/Application Exerciser Monkey. (2017). https://developer.android.com/studio/test/monkey.html.

[6] Android-x86. 2017. Android-x86 Open Source Project Announcement. (2017). http://www.android-x86.org/.

[7] Android-x86. 2017. Houdini Source Tree. (2017). https://sourceforge.net/p/android-x86/vendor_intel_houdini/ci/kitkat-x86/tree/.

[8] Appetize.io. 2017. Appetize.io. (2017). https://appetize.io/.

[9] Tanzirul Azim and Iulian Neamtiu. 2013. 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)*. ACM, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[10] Pedro Casas, Pierdomenico Fiadino, and Arian Bär. 2014. Understanding HTTP Traffic and CDN Behavior from the Eyes of a Mobile ISP. In *Proc. Passive and Active Measurement (PAM)*.

[11] Wontae Choi, George Necula, and Koushik Sen. 2013. 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)*. ACM, New York, NY, USA, 623–640. https://doi.org/10.1145/2509136.2509552

[12] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. 429–440. https://doi.org/10.1109/ASE.2015.89

[13] Collectd. 2017. The system statistics collection daemon. (2017). https://collectd.org/.

[14] Crowdflower. 2017. Crowdsourcing platform. (2017). https://www.crowdflower.com/.

[15] Telecommunication Networks Group Politecnico di Torino. 2017. Tstat website. (2017). http://tstat.polito.it.

[16] EMMA. 2017. EMMA: a free Java code coverage tool. (2017). http://emma.sourceforge.net/.

[17] Google. 2017. Firebase Test Lab for Android. (2017). https://firebase.google.com/docs/test-lab/.

[18] Google. 2017. HTTPS at Google – Google Transparency Report. (2017). https://www.google.com/transparencyreport/https.

[19] Grafana. 2017. The open platform for analytics and monitoring. (2017). https://grafana.com/.

[20] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. 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)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[21] Tobias Hossfeld, Christian Keimel, Matthias Hirth, Bruno Gardlo, Julian Habigt, Klaus Diepold, and Phuoc Tran-Gia. 2014. Best Practices for QoE Crowdtesting: QoE Assessment With Crowdsourcing. *Trans. Multi.* 16, 2 (Feb. 2014).

[22] Gael H. (Intel). 2013. Performance Results for Android Emulators - with and without Intel HAXM. (2013). https://goo.gl/D6rUf2.

[23] Maciej Korczynski and Andrzej Duda. 2014. Markov chain fingerprinting to classify encrypted traffic. In *Proc. IEEE INFOCOM*.

[24] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. 2016. 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)*. ACM, New York, NY, USA, 365–372. https://doi.org/10.1145/2987443.2987456

[25] Linux-KVM. 2017. Qcow2 image format. (2017). https://www.linux-kvm.org/page/Qcow2.

[26] ARC Welder Official Compatibility List. 2017. goo.gl/Q0fy3m. (2017).

[27] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[28] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[29] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. *arXiv preprint arXiv:1612.04433* (2016).

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

[31] Ashkan Nikravesh, Hongyi Yao, Shichang Xu, David Choffnes, and Z. Morley Mao. 2015. Mobilyzer: An Open Platform for Controllable Mobile Network Measurements. In *Proc. ACM MobiSys*. 389–404.

[32] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. 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)*. ACM, New York, NY, USA, Article 15, 6 pages. https://doi.org/10.1145/2766498.2766522

[33] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale. In *Proc. Network & Distributed System Security Symposium (NDSS)*.

[34] Puma. 2017. A Modern, Concurrent Web Server for Ruby. (2017). http://puma.io/.

[35] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. 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)*. ACM, New York, NY, USA, 361–374. https://doi.org/10.1145/2906388.2906392

[36] T Richardson and J Levine. 2011. The Remote Framebuffer Protocol. (2011). https://tools.ietf.org/html/rfc6143.

[37] Global Internet Phenomena Sandvine. 2016. Spotlight: Encrypted Internet Traffic. (2016). https://www.sandvine.com/trends/encryption.html.

[38] Raimondas Sasnauskas and John Regehr. 2014. 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)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/2632168.2632169

[39] Selenium. 2017. Web Browser Automation. (2017). http://www.seleniumhq.org/.

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

[41] Spice. 2017. Spice. (2017). https://www.spice-space.org/.

[42] Spice. 2017. Xspice. (2017). https://www.spice-space.org/xspice.html.

[43] Alok Tongaonkar, Shuaifu Dai, Antonio Nucci, and Dawn Song. 2013. Understanding Mobile App Usage Patterns Using In-app Advertisements. In *Proc. Passive and Active Measurement (PAM)*.

[44] UserTesting. 2017. User Experience Research Platform. (2017). https://www.usertesting.com/.

[45] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. https://doi.org/10.1145/2382756.2382797

[46] Matteo Varvello, Jeremy Blackburn, David Naylor, and Kostantina Papagiannaki. 2016. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 399–412.

[47] Google Chrome ARC Welder. 2017. https://developer.chrome.com/apps/getstarted_arc. (2017).

[48] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering*, Vittorio Cortellessa and DÃąniel VarrÃş (Eds.). Number 7793 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 250–265. http://link.springer.com/chapter/10.1007/978-3-642-37057-1_19 DOI: 10.1007/978-3-642-37057-1_19.

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

[50] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia (MoMM '13)*. ACM, New York, NY, USA, 68:68–68:74. https://doi.org/10.1145/2536853.2536881

[51] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger P. Yu, and Martin Abadi. 2012. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *Proc. Network & Distributed System Security Symposium (NDSS)*.