RANDR: Record and Replay for Android Applications via Targeted Runtime Instrumentation

Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse K. Coskun, Manuel Egele
Electrical and Computer Engineering Department, Boston University, Boston, MA, USA
{sahin, aliyevaa, hrhrnm, acoskun, megele}@bu.edu

Abstract— **The ability to repeat the execution of a program is a fundamental requirement in many areas of computing from computer system evaluation to software engineering. Reproducing executions of mobile apps, in particular, has proven difficult under real-life scenarios due to multiple sources of external inputs and interactive nature of the apps. Previous works that provide record/replay functionality for mobile apps are restricted to particular input sources (e.g., touchscreen events) and present deployment challenges due to intrusive modifications to the underlying software stack. Moreover, due to their reliance on record and replay of device specific events, the recorded executions cannot be reliably reproduced across different platforms.**

In this paper, we present a new practical approach, RANDR, for record and replay of Android applications. RANDR captures and replays multiple sources of input (i.e., UI and network) without requiring source code (OS or app), administrative device privileges, or any special platform support. RANDR achieves these qualities by instrumenting a select set of methods *at runtime* within an application's own sandbox. In addition, to enable portability of recorded executions across different platforms for replay, RANDR contextualizes UI events as interactions with particular UI components (e.g., a button) as opposed to relying on platform specific features (e.g., screen coordinates). We demonstrate RANDR's accurate cross-platform record and replay capabilities using over 30 real-world Android apps across a variety of platforms including emulators as well as commercial off-the-shelf mobile devices deployed in real life.

I. INTRODUCTION

Reproducibility of the execution of a program is a core capability necessary in many areas of computing: workload characterization for computer architecture [1], system software optimization for performance or energy [2], or software debugging and testing [3]. For instance, for software testing and quality assurance, it is often desirable to reproduce the high-coverage test cases obtained via human input [4]. Similarly, for OS-level power or performance optimization, the same execution of a program is often repeated under different scheduling or power management policies to explore energy and performance tradeoffs on real-life devices [5]. Meeting this core reproducibility requirement, however, has proven particularly challenging for mobile applications (i.e., apps) [6], [7]. Unlike traditional "run-to-completion" type applications with minimal interaction found in desktop/server systems, execution of mobile apps are heavily influenced by non-deterministic input sources such as user interactions, network, sensors, or random numbers.

Prior record and replay tools for desktop and server platforms based on low-level events (e.g., system calls [8], CPU instructions [9], [10]) are not readily useful for real-life mobile systems and apps due to their high overhead [6]. Beyond the need for reproducing multiple non-deterministic factors such as UI and network events, accurate record and replay of Android apps also require preserving the timing of events [11]. For instance, the timing between the UI events, the primary input source in mobile apps, is crucial to determine the type of the UI interaction (e.g., (long) click, swipe etc.) [6], [12].

While various approaches are specifically geared towards record and replay of mobile apps, several limitations impair their accuracy and practicality. First, most prior work [6], [7], [12], [13] rely on platform-specific raw screen coordinates of UI inputs which restricts the replay capabilities only to one specific device [13] or screen resolution [6]. Moreover, some approaches focus purely on UI events [12], [13], [14] and cannot handle execution variations that arise due to other common factors such as network or random number inputs. Second, many of the prior approaches incur practical drawbacks as they require access to the app source code [14], [15], [16] or require intrusive modifications to the underlying OS or virtual machine [6]. Most real-world apps in Android markets (e.g., Play Store) are closed-source. Modifying the OS is tedious to implement and maintain [7] due to the fragmented Android ecosystem with many software versions and custom devices [12] and require a device with an unlocked boot loader to install new images. Thus, existing approaches achieve limited applicability to real-life off-the-shelf mobile devices due to manufacturer imposed restrictions against unlocking boot loaders and gaining administrative privileges (i.e., *root*).

This paper proposes a record and replay system, RANDR, that provides cross-platform and timing sensitive record and replay of multiple sources of inputs without imposing the restrictions that undermine practicality. RANDR deterministically replays mobile apps by recording and replaying app inputs (i.e., UI, network, and random numbers) by *dynamically instrumenting* [1] a set of target methods within an app's own sandbox. RANDR solves the deployment challenges of existing tools [6], [12], [13] since it does not require any persistent modifications to the system that would need elevated privileges (i.e., RANDR only requires minuscule generic app-rewriting to trigger our dynamic instrumentation). Through *strategic* selection of the target instrumentation points, RANDR retains sufficient contextual information about inputs (e.g., the partic-

[1]We use the term hooking and dynamic instrumentation interchangeably.

ular UI widget a user interacts with) to enable cross-platform replayability without sacrificing generality.

We implement a prototype of RANDR for Android. RANDR hooks into a set of target Java methods to capture UI inputs as well as the random numbers that cause variant app behavior if not kept deterministic across record and replay. By intercepting the UI inputs in Java APIs within the Android framework, RANDR automatically associates UI events as interactions with a particular UI widget (i.e., *widget-sensitive*) irrespective of where these UI components appear precisely on screen, providing cross-platform replay capability. In addition, RANDR intercepts the network traffic at the system call layer by hooking into standard C libraries. Capturing and reproducing network traffic at the system call boundary gives RANDR generality to handle different network library implementations or network protocols. By accurately capturing these multiple data and event streams and performing timing sensitive and widget-sensitive replay of UI interactions, RANDR successfully reproduces the behavior of popular Android apps across different devices. We evaluate fidelity of RANDR qualitatively through the visual similarity between record and replay of over 30 apps as well as quantitatively by comparing the similarity of methods executed by each app during record and replay.

In summary, this paper makes the following contributions:

- We propose a dynamic instrumentation approach (§ III) for record and replay of mobile apps. Our approach effectively captures UI, network, and random number inputs within an apps's own sandbox without requiring any elevated privileges or persistent system modifications.
- We propose a widget-sensitive UI replay methodology that provides cross-platform replay capabilities (§ IV-B2).
- We propose a native library level record and replay methodology that can reproduce popular network protocols (e.g., `HTTP`, `HTTPS`, `FTP`) (§ IV-B3).
- We implement a prototype of RANDR. We show RANDR's replay success with over 30 real-world apps across different devices and Android versions (§ V).

II. BACKGROUND

This section briefly reviews the Android concepts that are relevant to our record and replay system. More specifically, we discuss how apps are distributed and executed on the Android platform. We also describe Android's input handling mechanism as well as the principles of dynamic instrumentation.

Android System. Android apps are mainly written in Java on top of the APIs provided by the Android framework, compiled into Dalvik bytecode and executed by the Android Runtime (`ART`). Android apps can also implement a part of their functionality in native code (e.g., in `C/C++`). These apps are distributed in the form of `APK` files that contain an app's bytecode (`.dex`), resources, assets, and a manifest file. The manifest file declares a set of permissions that grants the apps access to additional functionality (e.g., network, storage).

The Android platform is built upon a modified version of the Linux kernel and provides each application with an isolated execution sandbox. Each application runs in its own process and with its own instance of the `ART`. To provide low-level system functionalities to the apps, the Android framework consists of a set of native libraries (e.g., `libssl` for SSL support, `libc` for standard C library operations).

Android UI System and Input Handling. User interactions over the touchscreen are the primary source of input for Android apps. Android describes UI events using MotionEvent and KeyEvent classes, each extending the InputEvent class. MotionEvents specify the user input in terms of an action code (e.g., `ACTION_UP`, `ACTION_DOWN`) and screen coordinates. A sequence of MotionEvents can describe any user gesture such as long press, fling, and pinch. KeyEvents describe a key that has been pressed (e.g., volume, virtual keyboard).

To facilitate user interactions with an application, the Android framework provides a rich set of UI elements (*widgets*) such as Buttons, TextViews, ImageView, or ScrollView. Moreover, developers can implement custom widgets. To ensure compatibility with the Android framework, every widget must be derived from the View or ViewGroup class. Views are UI elements that represent interactive objects on the screen, while a ViewGroup is responsible for organizing Views and other ViewGroups into a layout tree. An application can chose to define its layout tree statically (i.e., at design time), dynamically (i.e., at runtime), or using a combination of both.

In Android, the Activity class describes a single screen of an app's UI. An Activity is given a rectangular area on the screen (i.e., a Window) on which to draw its user interface. All UI components in a Window form one layout tree hierarchy with the root view defined by the ViewRootImpl class. This class describes the behavior of the window and has two main functions relevant to RANDR. First, each ViewRootImpl class instance registers an InputEventReceiver, a low level mechanism to deliver all InputEvents to an app's window. Second, this class is responsible for traversing the view hierarchy to determine which View will receive user input.

Dynamically Instrumenting Java APIs in `ART`. Each `Java` class is internally represented by a `Class` object in memory in `ART`. A virtual method table (i.e., vtable) within a `Class` is used to resolve virtual methods and implement polymorphism. In `ART`, the `ArtMethod` structure corresponds to the internal C++ representation of both static and virtual Java methods. Thus, the vtable of a `Class` essentially holds pointers to `ArtMethod` objects in memory. Each `ArtMethod` object contains an *entry point* which essentially points to a code that performs the necessary set-up and clean-up, and executes the target method's code. Hooking in `ART` can be accomplished either by modifying an entry in the vtable to point to a different `ArtMethod` object [17] or by modifying the entry point within an `ArtMethod` to point to a different location [18]. The former approach allows to hook only into virtual methods. RANDR adopts the latter entry point hooking approach which works with both static and virtual methods.

The reflection API supported in the Java Native Interface (JNI) can be used to obtain a handle to `ArtMethod` object

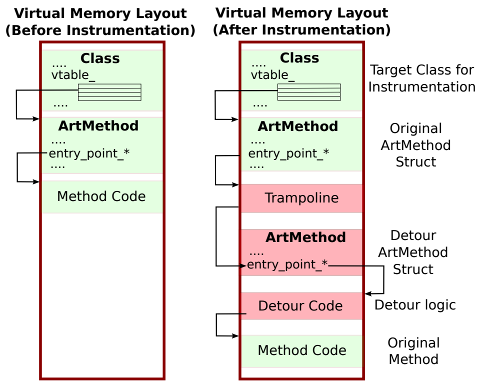


Fig. 1: Dynamic instrumentation in memory. Green regions represent the original class and method structures in memory while red regions highlights the dynamically injected components.

for a target Java method of interest. Once the address of the `ArtMethod` struct corresponding to the target method is known, the entry point can be accessed by a fixed offset and modified to point to a different address. Modifying the address within the entry point enables executing a *hook method* upon invocation of the target Java method. Once control is diverted to the hook function, the target method arguments can be recorded or modified. The original target method can also be invoked from the hook function to implement the original method behavior. Once the original target method is executed, the return value can also be recorded or modified as necessary. Figure 1 summarizes this process by illustrating the state of the virtual memory layout of an app before and after hooking.

III. RANDR OVERVIEW

The goal of RANDR (Figure 2) is to provide the capability to reproduce the executions of mobile apps recorded on one platform across different devices (i.e., cross-platform replay) in a practical manner. RANDR achieves practical merits by providing record and replay capabilities without requiring any administrative device privileges or source code modifications (i.e., OS or app). RANDR realizes these capabilities through a combination of static app rewriting and dynamic framework instrumentation that allows to capture and reproduce the inputs to a set of target method calls (i.e., both Java and C APIs) in the Android framework. This section serves as an overview of these static and dynamic instrumentation components. We also describe RANDR's cross-platform replay methodology along with an overview of the working principles of RANDR.

Dynamic instrumentation is the key component of RANDR that *hooks* into a set of target methods in memory to capture (i.e, for record) or modify (i.e., for replay) input arguments or return values. Our aim is to capture and reproduce the inputs to the app that originate from external non-deterministic sources such as UI interactions, network, and random numbers. We detail our specific instrumentation targets to handle these non-deterministic sources in § IV and proceed with working principles of RANDR during record and replay stages.

Record Stage (❷): During the recording phase, we direct the control flow from instrumented methods to custom *hook*

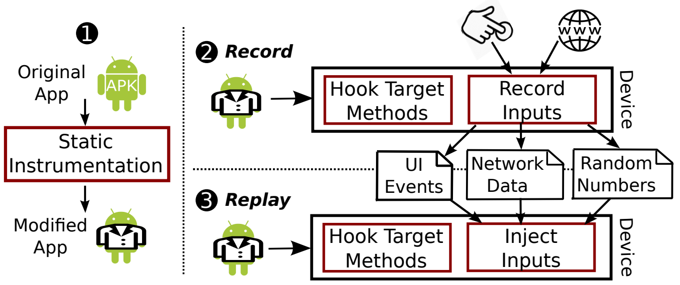


Fig. 2: Macro view of our RANDR record and replay approach.

methods that record inputs into a trace file. RANDR also records timestamps to ensure that the timing between the method invocations can be preserved during replay. For the case of user interactions, simply recording the fields of input arguments (e.g., raw screen coordinates from `InputEvent`) does not provide sufficient information to reproduce executions on a different platform where the positions of UI widgets can differ. RANDR addresses this problem by generating *stable* (i.e., platform-independent) identifiers for each UI widget and contextualizing each UI event as an interaction with respect to a particular UI widget. Simply recording the received network data also may not be sufficient for replay due to underlying cryptographic protocols establishing secure communication based on non-deterministic inputs (e.g., `TLS` session keys derived from external entropy sources [19]). To address this challenge, we study Android's `SSL` implementation to identify sources of non-determinism for network replay and determine the instrumentation points for enabling reproducibility of network sessions. § IV details how RANDR addresses these challenges concretely.

Replay Stage (❸): RANDR's replay logic differs depending on whether the target method invocation is initiated from the app or the Android runtime (i.e., a *callback*) while recording. For callbacks, RANDR directly invokes the original method with saved arguments for replay. For instance, UI events are asynchronously delivered to an app by the Android runtime during record (i.e., a callback) and RANDR directly invokes the necessary target methods during replay. RANDR adds sleep intervals between method invocations according to the recorded timestamps to preserve timing. For the method calls that originate from the app (e.g., a call to `Math.random()`), once the app invokes the target method, RANDR replaces the original method arguments with those from the trace file.

RANDR requires only minimal static instrumentation on the app (❶). The purpose of this static instrumentation stage is to trigger our dynamic framework instrumentation from within the app. RANDR decodes the app resources and `dex` files, rewrites application's `bytecode` such that the app loads and executes our dynamic instrumentation component during app launch. RANDR exclusively injects code into app-startup and does not rewrite any of the app's code beyond that. The output of this stage is a new instrumented `APK` file.

Overall, performing the framework instrumentation at runtime and from within the app provides RANDR with unique advantages. First, the target method structures can be identified and patched in memory at runtime without requiring any static OS modifications or access to any device software. Second, since target method structures are modified by the app in its own process address space, RANDR does not require any privileged mode of operation or root permissions. Therefore, RANDR is able to capture multiple input sources across different software layers to provide record and replay capabilities on unmodified Android devices.

IV. RANDR IMPLEMENTATION

This section describes our implementation of the RANDR system. § IV-A details the static application instrumentation necessary to trigger our dynamic instrumentation. In § IV-B, we describe the hooking mechanism along with how we leverage this mechanism to enable record and replay of UI, network, and random number inputs for Android apps.

A. Static Application Instrumentation

RANDR's static instrumentation component modifies the application resources and bytecode such that the app loads our dynamic instrumentation library (§ IV-B) at app-launch. We use `apktool` [20] to disassemble the bytecode from the application's `dex` files and obtain the manifest file (`AndroidManifest.xml`). The disassembled bytecode is in readable and editable `smali` [21] representation. To ensure our hooks are in-place before the app starts to execute, RANDR injects a small stub code into the `Application` subclass of the app. We choose this approach as the `Application` subclass is the first class to be instantiated for each application. The small stub code simply invokes `System.loadLibrary()` to load our native hooking library, which we insert into the `APK`. RANDR also modifies the manifest file to add storage access permissions to allow the application to read and write trace files.

B. Dynamic Runtime Instrumentation

This section details how RANDR realizes record and replay capabilities for Android apps by dynamically hooking into a set of target methods. We first describe the implementation of our hooking library in § IV-B1. RANDR intercepts UI and random number inputs by instrumenting a set of Java APIs as detailed in § IV-B2 and § IV-B4. RANDR achieves network record and replay by hooking at system call wrappers in `libc`, which provides generality to our framework (e.g., RANDR can seamlessly support third party network libraries or different protocols). § IV-B3 describes our network replay approach.

1) Hooking into Java and Native APIs: We build our Java API instrumentation setup on top of an open-source hooking framework for `ART` (i.e., `YAHFA` [22]). As per the entry point hooking process described in § II, `YAHFA` creates a backup of the original `ArtMethod` object and replaces the target of the entry point with a trampoline that simply jumps to the entry point of the `ArtMethod` object corresponding to our hook method. Performing the original method call behavior simply requires invoking the backed-up version of the original `ArtMethod` from our hook functions. During record, in addition to logging the method arguments and return values, RANDR invokes the original methods to allow the application to execute in its default behavior.

Our implementation of this hooking infrastructure is fully contained in a shared native library and leverages the reflection API support in JNI to obtain references to method instances at runtime (i.e., via `GetStaticMethodId()`). Our framework uses dynamic class loading capabilities provided by the Android framework (i.e., via `DexClassLoader`) to dynamically load hook methods as well as other replay components into the virtual address space of the target application. The hook methods are written in Java, which provides ease of use and makes RANDR easily extensible to add other instrumentation points as necessary.

Our implementation of the native library hooks uses the `AndHook` framework [23] which provides a hooking interface for various Application Binary Interfaces (`ABIs`) including armeabi-v7a, arm64-v8a, x86 and x86_64. `AndHook` locates the image of a given shared library in the process address space, parses the `ELF` structure to identify the address of the target functions we wish to instrument, and inserts necessary jumps at this address to realize method hooking. RANDR uses the `AndHook` framework to hook into bionic and crypto libraries for recording and replaying the network events.

2) Recording and Replaying UI Inputs: RANDR records and replays UI inputs by hooking into a set of Java APIs in the Android framework. Performing the interception at the framework layer provides RANDR with the generic UI input capture capabilities while retaining sufficient contextual information in the Android framework to associate these events with their target UI widgets. By recording and replaying UI events with respect to their target widgets, RANDR achieves cross-platform reproducibility of UI events independent of where these widget appear on a particular device screen.

RANDR needs to uniquely identify widgets in an application in a manner that is stable and consistent across multiple application runs and across different devices. To address this problem, RANDR first hooks into the `onDraw()` and `draw()` methods in the `View` class. This allows RANDR to monitor all widgets that appear on the screen before the users can interact with them and enables RANDR to work with both statically defined and dynamically created widgets. As a second step, RANDR assigns each widget a stable identifier that is derived using a combination of widget properties. More specifically, RANDR collects information such as the private fields of the `View` class instances that describe whether a widget is enabled, will draw or scrollable. We found that other properties that can be used for widget identification such as focusableness and clickability are often changed at runtime and, are thus, not suitable for cross-platform widget identification. In addition to the `View` class properties, RANDR obtains any text displayed by a widget. Finally, RANDR uses the position of the widget in the UI tree. Since the full path to the widget's node in the UI layout (i.e., the XPath)

TABLE I: Main target Java instrumentation points for UI replay used in RANDR during record (left) and replay (right).

Record	**Replay**
View.onDraw()	
View.draw()	
InputEventReceiver.dispatchInputEvent()	
View.onTouchEvent()	Activity.onAttachedToWindow()
ViewGroup.dispatchTouchEvent()	
View.onAttachToWindow()	

can change across different devices, RANDR only uses the relative position of a widget. We empirically established that acquiring a widget's 5 ancestors nodes and their classnames in combination with other widget properties described above is sufficient to generate cross-platform stable identifiers.

During recording, RANDR intercepts the MotionEvents received by a widget and records the widget's identifier, MotionEvent action, timestamp and the relative position of the MotionEvent within a widget. RANDR also registers listeners to every `Editable` widget to record any entered text.

During replay, RANDR first finds the location of the target widget on the screen. RANDR generates MotionEvents with coordinates adjusted to the new position and size of the identified UI widget on the screen. To inject newly generated MotionEvents, RANDR keeps track of currently active ViewRootImpl class instances that are responsible for handling and managing View hierarchies in a window presented to a user. RANDR injects the MotionEvents back into the windows that the identified view resides in the order corresponding to event timestamps. By replaying the UI interactions through generation of adjusted MotionEvents, RANDR lets the Android framework handle the invocation of correct callback methods depending on the type of the UI widgets. To keep the app responsive during replay, RANDR sends the UI events from a dedicated background thread. Table I summarizes RANDR's instrumentation points in Java APIs for UI record and replay.

RANDR includes cross-device support for the scrolling gesture as this gesture is often required to explore an application. We implemented scrolling using the following method. During recording, we log the timestamp, the identifier of the scrollable widget as well as the direction of the events. Moreover, we identify the last visible widget to the user at the moment when the scrolling gesture ended. During replay, we simply inject scrolling events into the recorded scrollable widget until the recorded (last) visible widget is drawn on the screen.

3) Recording and Replaying Network I/O: RANDR intercepts network communications by hooking into the system call wrapper methods in Android's standard C library (`libc`) implementation called `Bionic`. By performing the record and replay at the native system call wrapper layer, RANDR is agnostic to different network libraries (e.g., OkHttp, Volley or Picasso) or protocols (e.g., `HTTP`, `HTTPS` or `FTP`).

Unlike protocols such as `HTTP` and `FTP`, recording and replaying the results of system calls is not sufficient to reproduce `HTTPS` traffic, which is widely popular among apps. The challenge stems from the underlying cryptographic protocols (e.g., `TLS`) used in `HTTPS` which establish new cryptographic keys for each network session. Thus, it is not possible reproduce network sessions by simply replaying the recorded encrypted content through system calls as the data cannot be decrypted with the new session keys generated during replay. Our key insight for tackling this problem is that the root cause of non-determinism is due to the client- and server-side random numbers used for generating the session keys during the `TLS` handshake. RANDR inherently captures the server-side random inputs by intercepting the system calls. Our observation is that the client (i.e., the Android device that RANDR runs on) receives the server-side random numbers via the system calls (e.g., `read()`) which RANDR records and replays. To handle the client-side random numbers, RANDR hooks into the cryptographic libraries. Specifically, during both record and replay, RANDR fills the buffer argument of the `RAND_bytes()` method in the `libcrypto` library containing random data with a fixed sequence of bytes.

TABLE II: Target native instrumentation points in RANDR.

Instrumented Lib	**Instrumentation Points**
libc.so	*socket(), connect(), read(), write(), close(), poll(), sendto(), recvfrom(), shutdown()*
libcrypto.so	*RAND_bytes()*

Table II provides the specific set of methods RANDR instruments for network record and replay. The `libc` library contains the wrapper methods for system calls. We determined the set of system call wrappers listed in Table II by the following approach. First, we manually analyzed the system calls to identify the ones relevant for networking. Next, to identify which of these network related system calls are used in Android apps, we analyzed a set of system calls that are collected (i.e., via `strace`) by exercising 400 real-world apps with Android's monkey exerciser [24].

During record, once a `connect()` call is made on a socket, we generate a trace file associated with the target address and log the arguments and return values of the subsequent system calls (e.g., `read()`, `recvfrom()`) on this socket to the generated trace file. We extract the target address from the `sockaddr` type argument of the `connect()` method. The target address corresponds to an `IP` address for `AF_INET` or `AF_INET6` type sockets while corresponding to a local file path for `AF_UNIX` type local sockets. RANDR also invokes the original methods to obtain and record the return values.

During replay, once a `connect()` method is called on a socket, RANDR finds the trace file associated with the target address and subsequent system calls on this socket retrieve their data from the corresponding trace file. For instance, during replay, `read()` system calls fills data into its `buf` argument from the recorded bytes in our trace files, as opposed to fetching data over the network. Due to multiple potential connections to the same address during record, RANDR may need to disambiguate the trace logs to read from.

For such cases, RANDR relies on the ordering between the `connect()` system calls.

An application may attempt to fetch data from the same address through multiple connections from different threads. Since RANDR may not be able to disambiguate the connections between such threads whose execution order may vary between runs, we limit the level of concurrency during record and replay. Specifically, RANDR hooks into the constructor, `setMaximumPoolSize()` and `setCorePoolSize()` methods of the `ThreadPoolExecutor` class to ensure that the size of a thread pool does not exceed one. Note that enforcing a schedule order between threads requires modifications to the kernel [25] which would change the nature of RANDR.

RANDR is also able to reproduce DNS resolutions by recording and replaying the system calls to a local DNS deamon (i.e., `\dev\socket\dnsproxyd`). For DNS resolutions, RANDR uses the target host name to identify the correct log file to replay from. The target host name is sent to the DNS deamon through a local socket using a `write()` system call. Thus, to obtain the target host name for a DNS resolution, RANDR uses the `buf` argument of the `write()` system call to the local DNS deamon.

4) Recording and Replaying Random Numbers: Random numbers present another significant source of non-determinism in Android apps which, if not captured and replayed, can lead to different app behaviors [26]. We note that such random input dependent variations could even be intentional from the developer's perspective (e.g., 2048 game app).

RANDR hooks into the pseudo-random number generator in the Java API. Specifically, we record the return values from the `next()` method in the `java.util.Random` class. We choose this specific instrumentation point since it is the common subroutine among other public random number APIs (e.g., `Random.nextInt()`, `Random.nextBytes()`, `Math.random()`). During replay, return values of the `next()` method is overridden by the next number in the sequence of recorded values.

V. EVALUATION

This section provides our evaluation of RANDR. Specifically, we seek to answer the following research questions.

RQ1 Does RANDR accurately capture UI, network, and random number inputs and achieve cross-platform replayability of real-world apps?

RQ2 Does RANDR's generic network replay approach handle different network protocols?

RQ3 How much effort is needed to port RANDR to different platforms?

RQ4 Does RANDR incur slowdown on apps that may disrupt the replay accuracy?

We first describe our evaluation setup in § V-A. § V-B addresses **RQ1** and **RQ2** by describing our fidelity metrics and assessing RANDR's replay capabilities. We also highlight the portability of our approach (**RQ3**) by evaluating RANDR across different Android versions and quantifying the changes needed to support a new platform. § V-C evaluates RANDR's overhead to verify the practical nature of our approach (**RQ4**). In § V-D, we illustrate two case studies that highlight the importance of handling multiple non-deterministic inputs and the benefits of accurate replay for workload characterization.

A. Evaluation Setup

Our experimental setup for evaluation consists of devices with diverse characteristics (e.g., emulator, real device, different software versions, and screen resolutions). Our aim is to highlight the ease of deploying RANDR across different platforms and assessing its cross-platform replay capabilities. Specifically, we evaluate RANDR on a real Pixel XL smartphone (running Android 8.0) and two Android x86-64 emulators with different device specifications: Nexus 4 and Pixel 2 XL running Android 8.1 and 8.0, respectively. We choose these specific emulators due to a significant difference in their screen resolution (i.e., 768x1280 `px` for the Nexus 4, 1440x2880 `px` for the Pixel 2XL).

B. Fidelity of RANDR

This section evaluates the replay capabilities of RANDR. First, we provide our methodology for establishing the replay accuracy. Next, we evaluate RANDR's replay abilities for both UI and network events. Finally, we show RANDR's ease of portability to real devices and different Android versions.

Fidelity Metrics: We use two metrics to evaluate the accuracy of our replay using RANDR with respect to the original execution during record. First, as in much of the prior work [6], [7], [13], [14], we use the user visible visual states of the app as a proxy for replay accuracy. Due to the subjective nature of this visual similarity comparison, we also use the Jaccard similarity between the set of methods executed during record and replay. This Jaccard similarity metric provides us with a finer granularity (in comparison to visual output similarity) way to quantify RANDR's replay accuracy. We modified the `ART` source code [27] in both emulators to extract the set of executed methods. Note that these modifications to `ART` are only used to assess the fidelity of RANDR and are not necessary to use or deploy RANDR.

Cross-Device UI Replay with RANDR: To test RANDR's cross-platform UI replay capabilities, for each app, we record UI inputs with RANDR on the Nexus 4 emulator and replayed the recorded traces on both Nexus 4 and Pixel 2 XL emulators. We exercise each app for approximately 90 seconds manually to evaluate RANDR's ability to reproduce realistic real-life app interactions not provided by randomized UI exercisers (e.g., Android Monkey) [24]. We re-install the app before each execution since we also want to test whether RANDR's replay functionality is self-contained. Note that RANDR's network replay functionality is disabled for this particular experiment.

To show RANDR's cross-platform UI replay capabilities on real-world closed-source Android apps, we collect a set of the most popular apps from most Google Play Store categories. From each category, we pick the apps with the highest number of downloads according to AppBrain rankings [28] that

TABLE III: The set of apps used for evaluating RANDR's cross-platform UI replay capabilities.

Category	**App Name**	**#Downloads**	**Scrolling**	**Visual Success (Nexus 4)**	**Visual Success (Pixel 2XL)**	**Jaccard Similarity (Nexus 4)**	**Jaccard Similarity (Pixel 2XL)**
Entertainment	Mi Video	100M+	no	✓	✓	98.62%	98.74%
Finance	Currency Converter	10M+	no	✓	✓	98.34%	98.38%
Parenting	Baby Tracker	10M+	no	✓	✓	97.99%	97.71%
Comics	Draw Cartoons 2	5M+	no	✓	✓	97.43%	97.43%
House & Home	Zillow	10M+	yes	✓	✓	96.56%	91.00%
Maps & Navigation	Kakao Bus	10M+	no	✓	✓	97.82%	97.97%
Travel & Local	Where is my Train?	50M+	yes	✗	✗	69.96%	69.94%
Weather	The Weather Channel	100M+	no	✓	✓	96.99%	96.60%
Art & Design	Sketch	100M+	yes	✓	✓	99.08%	98.70%
Beauty	Mirror Camera	10M+	yes	✓	✓	98.12%	95.40%
Medical	Periods and Ovulation Tracker	100M+	no	✓	✓	95.17%	96.72%
Shopping	Wish	100M+	no	✗	✗	96.56%	96.02%
Sports	Onefootball	10M+	yes	✓	✓	99.23%	96.95%
Books & References	Oxford Dictionary of English	50M+	no	✓	✓	96.56%	98.07%
Business	OfficeSuite	100M+	yes	✗	✗	91.01%	84.21%
Communication	Messages	100M+	no	✓	✓	95.86%	94.76%
Dating	Hot or Not	10M+	yes	✗	✗	90.52%	90.68%
Education	DuoLingo	100M+	yes	✓	✓	99.97%	99.51%
Events	StubHub	5M+	no	✓	✓	98.90%	98.75%
Food & Drink	FourSquare City Guide	10M+	no	✓	✓	99.23%	99.15%
Health & Fitness	MyFitnessPal	50M+	yes	✗	✗	96.90%	99.08 %
LifeStyle	Daily Zodiac Horoscope and Astrology	10M+	no	✓	✓	97.90%	97.80%
Music & Audio	Amazon Music	100M+	yes	✗	✗	46.55%	46.58%
News & Magazines	Flipboard	500M+	yes	✓	✓	96.97%	94.66%
Personalization	Ringtone Maker	50M+	no	✓	✓	97.95%	98.06%
Photography	Zero Launcher	50M+	no	✓	✓	97.58%	97.53%
Productivity	MyJio	100M+	yes	✗	✗	97.54%	97.95%
Social	Kakao Story	50M+	yes	✗	✗	65.81%	66.13%
Tools	Clock	100M+	no	✓	✓	98.75%	98.69%
Video Players & Editors	Dubsmash	100M+	no	✗	✗	96.76%	95.93%

RANDR can successfully instrument (i.e., some apps perform tampering detection), that can execute in our emulators. We excluded five categories (i.e., Wear OS, Daydream, Auto and Vehicles, Games, and Libraries and Demo) for several reasons. Apps from the Wear OS category require an external device to operate, while the apps from the Daydream, Auto and Vehicles, and Games categories often render their UI components in native code. RANDR focuses on cross-platform replay of apps that use Android's UI toolkit. Apps in Libraries and Demo category often provide system services and do not provide a GUI to interact with the user. Table III lists the apps we collected, whether we used scrolling to explore the apps's functionality along with our results for visual and method set similarities on both emulators.

During our experiment, RANDR achieved visual similarity for 21 of the 30 apps. The failure in the visual similarity for the 9 remaining apps stems from different reasons. RANDR failed to replay 2 apps due to an interaction with a customized view, and 3 apps due to the presence of a WebView (see § VI). We do not report visual success for 4 apps (i.e., *Dubsmash, MyFitnessPal, Wish, HotOrNot*) because their user content is determined by a server, and is not consistent between record and replay. For example, RANDR was functionally able to replay *HotOrNot* and *MyFitnessPal* correctly. However, during replay on different devices, the *HotOrNot* app suggested the profiles of different users. The *MyFitnessPal* app retrieved several selected items entered during recording and saved in our associated app account. In the *Wish* app, the remote server tailors suggestions based on the information entered during recording, hence breaking the replay visually and functionally.

The last two columns in Table III show the Jaccard similarity between the set of methods recorded on the Nexus 4 emulator and the set of methods replayed on both the Nexus 4 and Pixel 2XL emulators. RANDR achieves 93.54% and 92.93% average Jaccard similarity on the Nexus 4 and Pixel 2XL emulators, respectively. We report high Jaccard similarities for all visually successfully replayed apps. For the apps that RANDR failed to replay functionally (i.e., due to a custom view or a WebView), we still observe high overall Jaccard similarities (e.g. *OfficeSuite, MyJio*), since the replay only diverged for a small portion of the execution. We find that subtle differences between record and replay in the Jaccard similarities for successfully replayed apps mainly stem from the following sources. First, RANDR's logic and instrumentation points for record and replay are not identical, hence different paths through the same code are executed. Second, we also do not always inject the same amount of scrolling events into both emulators, depending on whether the emulator already displays the views we are interested in (see § IV). This affects whether the app's response to scrolling is triggered (e.g. through *Gesture Detector or View.OnScrollChangeListener*). Note that the exact quantity of deviation in Jaccard similarities changes across the apps also due to the differences in the nature and positions of the widgets the user interacts with during the recording stage, triggering different execution paths in RANDR's implementation.

TABLE IV: The list of apps with network dependent functionalities and the different protocols used by each app. "#Sessions" reports the number of network sessions. A session starts and terminates when a socket is opened and closed, respectively.

Application	#Methods	HTTP	HTTPS	FTP	#Sessions	#Recorded System Calls	Network Data Volume (KB)	Visual Success (Nexus 4)	Visual Success (Pixel 2XL)	Jaccard Similarity (Nexus 4)	Jaccard Similarity (Pixel 2XL)
TripSit	10143	✓	✗	✗	4	11	8.28	✓	✓	96.63%	96.73%
My FTP Client-FTP Server Manager	24855	✗	✓	✓	53	888	638.75	✓	✓	92.82%	92.65%
MirrorCamera	25891	✗	✓	✗	48	920	541.25	✓	✓	95.12%	93.91%
Anecdote	17259	✓	✓	✗	17	200	17.26	✓	✓	95.58%	93.20%
eBooks	12563	✗	✓	✗	7	656	445.47	✓	✓	95.57%	95.44%
Transports Rennes	12208	✓	✓	✗	83	1237	814.84	✓	✓	94.42 %	94.47 %
BusTO	12315	✓	✗	✗	15	197	1174.17	✓	✓	97.13%	96.49%
WorldWeather	13863	✓	✗	✗	11	42	67.30	✓	✓	97.46%	96.75%
OpenManga	14958	✓	✗	✗	14	186	878.63	✓	✓	98.96%	97.89%
LeMonde	16413	✗	✓	✗	4	681	1633.99	✓	✓	96.91%	96.15%

Reproducing Network Sessions with RANDR: RANDR provides a generic approach for replaying network traffic within the apps via system call layer interception, and thus, can complement cases where the success of UI replay relies on the network data. To evaluate our current prototype implementation, we select apps that 1) use different protocols (i.e., `HTTP`, `HTTPS`, `FTP`) to show the generality of RANDR to different application-layer network implementations; 2) possess network-dependent functionalities. Note that our current implementation of RANDR cannot replay apps that use various features such as WebViews and multi-threading as described in § VI. Thus, we manually analyzed the presence of these features in apps retrieved from the Play Store and F-droid markets [29] to determine the set of apps described in Table IV. We also list the number of system calls and the total volume of network data RANDR records for each app. Identical to the previous section, we also report the Jaccard similarities.

Our native system call layer interception along with the capability to capture the random numbers used to generate the `TLS` session keys allows RANDR to reproduce network traffic that use different common network protocols (i.e., `HTTP`, `HTTPS` and `FTP` as shown in Table IV). Similar to the majority of the cases in our UI replay evaluation (i.e., Table III), RANDR achieves accurate replay across both emulators as indicated (Table IV) by the high method similarity as well as the visual similarity between the record and replay stages.

Portability of RANDR: RANDR is highly generic and can be easily ported to different platforms. For instance, RANDR can be deployed on unmodified off-the-shelf devices and requires only marginal changes to support different Android framework versions. To evaluate RANDR in this respect, we conducted an experiment where we recorded 10 random apps from our PlayStore dataset (Table III) on a Nexus 4 emulator running Android 8.1, and replayed on a commercial Pixel XL smartphone running Android 8.0. RANDR successfully reproduced all the executions. Note that we cannot report Jaccard similarity as the device runs an unmodified `ART` (i.e., not modified for method profiling). To add the required support for Android 8.1 in RANDR, we modified only 9 (2 in Java, 7 in C) lines of code due to the differences in method signatures.

C. *Overhead Evaluation*

We perform our overhead characterization with a custom app with timing functionalities and two basic features: (1) a button to measure the overhead of our instrumentation at Java APIs; (2) fetching the contents of a given `URL` to measure the overhead of our native layer instrumentation. Due to inherent measurement variations, we repeat our experiment consisting of fetching `craigslist.com` upon a button click 20 times. We experiment on the Pixel XL smartphone (Android 8.0).

The primary performance overhead of RANDR stems from the dynamic framework instrumentation performed during the app launch. We measure the duration of this *one-time instrumentation* to be 64.35 ms which is negligible as compared to app launch times (i.e., orders of seconds). Next, we measure any impact of RANDR on the latency of UI events which we quantify by measuring the time difference between the `dispatchInputEvent` (invoked upon click) and the `onClick` (invoked when the event is delivered to the app) methods. To quantify the overhead of RANDR's interception at the native layer, we also measure the time to read the content from the given `URL`. We measure the latencies when RANDR is both disabled (i.e., RANDR_OFF) and enabled (i.e., RANDR_ON) during recording. As shown in Figure 3, RANDR does not introduce any significant measurable performance overhead. We also quantify that RANDR's static instrumentation increases the size of `APK` files by 864K.

D. *Case Studies*

This section provides specific case studies to show several use cases and benefits of our RANDR system.

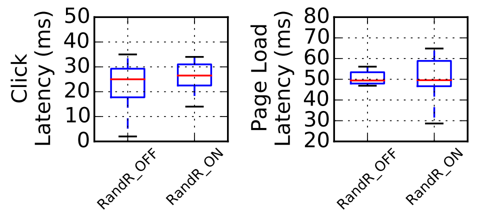
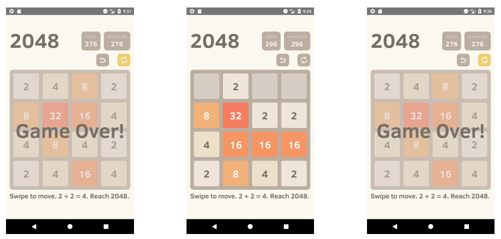


Fig. 3: Evaluating the performance impact of RANDR.



(a) Recorded (b) RERAN (c) RANDR

Fig. 4: Comparison of Reran [13] to RandR for 2048 app.

Behavior Variation due to Random Numbers: We demonstrate the importance of handling non-deterministic input sources other than touchscreen (i.e., random numbers) events with a specific case study using the 2048 game. We choose this specific app for illustration due to its popularity (+1M downloads in PlayStore) and randomized nature. We record a session of the game using both Reran [13] and RANDR where the game reaches the "Game Over" state and renders a new text on the screen as shown in Figure 4a. UI-only replay (i.e., using Reran) fails to reproduce the same state due to randomized location of numbers during replay. RANDR replays the same sequence of random numbers as in recording and, as seen in Figure 4c, the replay reaches the correct final state.

Impact of Accurate Replay on Performance Measurements: RANDR enables real-life experimentations on mobile systems with off-the-shelf mobile apps and, thus, can be used in power or performance characterization studies [1], [30]. In such a use case, RANDR can substantially improve the quality of measurement by reducing the execution variations caused by non-deterministic input sources. In Figure 5, we show the significance of replaying network traffic (i.e., using RANDR), as opposed to UI-only replay [13], [14]. In our specific case study, we measured the latency distribution over 20 executions of fetching a web page whose content changes over time (i.e., `thefakenewsgenerator.com`) using our custom app described in § V-C. RANDR replays the same recorded content each time (as opposed to fetching new data over the network) and, thus, is not effected by the network speed fluctuations or the web content changes on the server. This allows RANDR to significantly reduce the measurement variations and enable accurate characterization of mobile apps. Note that BBench [31], a popular browser benchmark suite, provides recorded contents of various websites for accurate workload analysis. RANDR, however, provides such analysis capabilities in a more generic way for a wider set of apps with network-dependent functionalities by automatically recording and replaying UI and network inputs under real-life use cases.

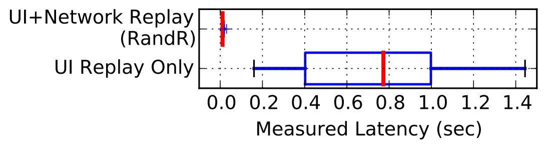


Fig. 5: Performance variance with and without network replay.

VI. LIMITATIONS AND FUTURE WORK

In this section, we state the limitations of our current implementation of RANDR and discuss potential improvements.

RANDR currently does not support record and replay of various inputs (e.g., GPS, sensor or audio data) or system events (e.g., `Broadcast`) that may also influence the program behavior. This limitation can be addressed by identifying the correct instrumentation points in the Android framework (e.g., in `LocationManager` and `SensorManager` classes) and providing RANDR with the necessary stubs for logging (for record) and overriding (for replay) input arguments.

Although RANDR supports complex UI gestures triggered with one finger (e.g., long-press or swipes), it does not provide cross-platform replayability of complex multi-finger gestures (e.g., pinch, zoom). Since the particular sizes of UI elements before and after such gestures are highly device specific, cross-device replay of such inputs is a challenging problem. Similarly, RANDR does not provide cross-platform replay for apps that perform customized rendering (commonly in games) as opposed to using Android's UI toolkit (i.e., using widgets) as RANDR cannot associate UI inputs with any widget. RANDR can, however, replay the UI events based on coordinates on the same device (as in most prior work [6], [7], [13]) with only small modifications to disable the widget sensitive replay and use raw MotionEvents. As RANDR relies on the timestamp of UI events, significant distortions in the timing behavior of an app (e.g., due to slower device) may impair RANDR's fidelity. In addition, the apps that override the Android Framework methods that RANDR uses for monitoring the user interactions and do not call the original supertype method, may not be successfully replayed. This limitation can also be addressed by identifying other instrumentation points in the Android framework to intercept the UI events.

In our experience developing RANDR, we have faced challenges in replaying apps that make use of `WebView` objects. As the contents of a `WebView` are rendered from a `WebView` system component (commonly, Google Chrome), RANDR cannot identify the UI components within the these objects. We have also experienced that RANDR may not be able to replay the network traffic from `WebView`'s rendering engine due to further instrumentation necessary to ensure deterministic `TLS` session keys. Addressing this challenge requires further effort to dissect default rendering engine's architecture.

Finally, to handle the non-determinism due to the ordering of thread executions, RANDR currently limits the number of threads per `ThreadPoolExecutor` to only 1 (§ IV-B3). However, the ordering between the threads across different `ThreadPoolExecutor` instances can still be non-deterministic. We also find that limiting the size of thread pools can cause some apps to hang due to long running tasks. This problem may be resolved in the future by establishing unique identifiers for the recorded network traces.

VII. RELATED WORK

The techniques we leverage in RANDR to achieve the novel capability of device-independent zero-privilege record and replay are related to works in two broad categories: GUI testing and exploration, and record and replay tools. This section briefly surveys the key aspects of these tools and describes how RANDR distinguishes itself from them.

GUI testing and exploration. Various previous work in academia [32], [33], [34], [35], [36] uses a combination of various widget properties (e.g., clickable or non-clickable) and/or their position in the UI hierarchy to uniquely identify widgets during the execution. Unlike these tools, RANDR recognizes UI widgets across different platforms and provides record and replay capability. Moreover, since RANDR monitors all widgets that appear on the device screen at runtime, it generically handles both statically defined and dynamically created widgets as opposed to various other tools [32], [33], [35] that achieve limited accuracy [14] due to reliance on Android's accessibility framework. Another UI testing tool, Appflow[37], synthesizes UI tests that are reusable across devices with different screen resolutions. While Appflow leverages machine learning to accurately identify widgets based on the set of features similar to RANDR, the tool still requires a developer to manually write customized UI flows for every app. RANDR only requires a user to naturally exercise an app.

We also compare RANDR against open-source frameworks that are frequently used by developers to test an app's UI on different platforms. Tools such as Robotium [38] and Espresso [15] instrument the source code of an app under test to monitor its execution and lifecycle. These tools are widget-sensitive and provide developers with fine-grained functionality such as specifying the interactions on widgets as well as the expected runtime behavior. UIAutomator [16] allows developers to monitor closed-source apps and track the interactions between the app and the system or other apps. While these automation tools do not require Android OS modifications, they do require manual effort as developers need to extract the GUI hierarchy and create testing scripts. RandR, on the contrary, can automatically record user interactions with an application, thus reducing the cost of writing tests, and does not require access to an application's source code to operate.

Record and Replay tools. RANDR also provides advantages over existing record and replay tools for Android. Automated record and replay systems such as Reran [13], Valera [6] and MobiPlay[7] rely purely on screen coordinates of user interactions. Since the coordinates of UI elements in an app often change according to screen resolution of a device, these tools are not suitable for cross-device replay. Barista [14] aims to provide an accurate cross-platform replay. However, due to its reliance on the Espresso framework, the recorded traces can only be integrated with open-source apps. In addition, the designs of RERAN, MobiPlay and Barista do not account for non-deterministic inputs beyond UI such as network and random number inputs. RANDR captures these additional input sources and provides an automated widget-sensitive replay capability across different devices for both closed- and open-source apps. While Valera also supports recording network traffic, its design allows the tool to only intercept `HTTP` and `HTTPs` requests from Android's `OkHTTP` library. RANDR provides a generic network record and replay approach based on native library instrumentation. Thus, RANDR is agnostic to different network libraries or protocols (e.g., `FTP`). RANDR also does not require any additional configuration on the device (e.g., a Man-In-The-Middle proxy [39]).

TABLE V: Comparison of Android test and replay frameworks.

	Robotium [38]	Espresso [15]	UI Automator [16]	AppFlow [37]	RERAN [13]	Valera [6]	Barista [14]	MobiPlay [7]	**RANDR**
Automated record & replay	✗	✗	✗	✗	✓	✓	✗	✗	✓
No root privilege	✓	✓	✓	✓	✗	✗	✓	✓	✓
No custom OS	✓	✓	✓	✓	✓	✗	✓	✓	✓
Support closed-source app	✗	✗	✗	✗	✓	✓	✗	✓	✓
Replay randomized data	✗	✗	✗	✗	✗	✓	✗	✗	✓
Replay network data	✗	✗	✗	✗	✗	✓	✗	✗	✓
Cross-platform replay	✓	✓	✓	✓	✗	✗	✓	✗	✓

The record and replay systems described above also exhibit certain practical drawbacks. While Valera require static modifications to an Android system image, Reran and MobiPlay (when handling closed-source applications) require root privileges to operate. Thus, to employ these tools, one needs to unlock the bootloader on the device. Unlocking can lead to system errors, security vulnerabilities [40] and is generally not possible on some devices. Moreover, any unverified modifications to the system components or file system can also lead to a non-functional device state. RANDR works on unmodified devices without any special privileges by relying on dynamic modifications to the Android software stack confined within the app process. Thus, RANDR provides non-intrusive record and replay capability and can be easily deployed across different Android versions and devices.

Table V summarizes the comparison of RANDR to existing approaches considering the desirable features of a record and replay system as described by Lam et al. [11].

VIII. CONCLUSION

This paper proposed the RANDR system to record and replay multiple non-deterministic sources of inputs in mobile apps in a practical and easy to use manner. RANDR captures the inputs to an app by hooking into a specific set of instrumentation points in the Android framework and native libraries. By performing the instrumentation within an app's own sandbox, RANDR runs on unmodified devices and works with real-world closed-source apps with only minimal modifications to the app's bytecode. RANDR captures UI and random number inputs by hooking into a set of Java APIs in the Android framework and provides widget-sensitive cross-platform replay capabilities. RANDR also reproduces network traffic in apps by intercepting the random inputs in the `TLS` protocol as well as the system call wrappers methods in `libc`. RANDR successfully replayed over 30 real-world Android apps across different devices and Android versions.

References

[1] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2013.

[2] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated cpu-gpu power management for 3d mobile games," in *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[3] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," pp. 641–660, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[4] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. Piscataway, NJ, USA: IEEE Press, 2017, pp. 16–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155569

[5] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 64–76.

[6] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA. New York, NY, USA: ACM, 2015, pp. 349–366. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814320

[7] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE. New York, NY, USA: ACM, 2016, pp. 571–582. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884854

[8] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *USENIX Annual Technical Conference (USENIX ATC)*. Santa Clara, CA: USENIX Association, 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[9] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772958

[10] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA. New York, NY, USA: ACM, 2003, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/859618.859633

[11] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and replay for android: Are we there yet in industrial cases?" in *Foundations of Software Engineering*, ser. ESEC/FSE. New York, NY, USA: ACM, 2017, pp. 854–859. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117769

[12] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 215–224.

[13] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 72–81. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486799

[14] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 149–160.

[15] "Espresso," https://developer.android.com/training/testing/espresso/.

[16] "Android ui automator," https://developer.android.com/training/testing/ui-automator.

[17] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime," in *IMPS@ESSoS*, 2016.

[18] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with tiro," in *Proceedings of the 27th USENIX Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2018, pp. 1247–1262. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277297

[19] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, 2013, pp. 659–668. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516706

[20] "Smali/baksmali," https://ibotpeaches.github.io/Apktool/.

[21] "Smali/baksmali," https://github.com/JesusFreke/smali.

[22] "Yahfa," https://github.com/rk700/YAHFA.

[23] "Andhook," https://github.com/asLody/AndHook.

[24] "Android ui exerciser monkey," https://developer.android.com/studio/test/monkey.

[25] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950370

[26] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *24th Annual Network and Distributed System Security Symposium (NDSS)*, March 2017.

[27] "Android open source project," https://source.android.com.

[28] "Appbrain," https://www.appbrain.com/stats/.

[29] "F-droid repository," https://f-droid.org/en/.

[30] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 45–54.

[31] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 81–90.

[32] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: ACM, 2018, pp. 2–8. [Online]. Available: http://doi.acm.org/10.1145/3278186.3278187

[33] Y. M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," *International Conference on Automated Software Engineering (ASE)*, pp. 238–249, 2016.

[34] F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, "Frauddroid: Automated ad fraud detection for android apps," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 257–268. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236045

[35] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of gui design violations for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 165–175. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180246

[36] M. Fazzini, M. Prammer, M. d&#39;Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 141–152. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213869

[37] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," 10 2018, pp. 269–282.

[38] "Robotium," https://github.com/RobotiumTech/robotium.

[39] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate Record-and-Replay for HTTP," in *USENIX Annual Technical Conference*, July 2015.

[40] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: On the security of bootloaders in mobile devices," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini