Systematically Testing Background Services of Mobile Apps

Li Lyna Zhang[⋆‡], Chieh-Jan Mike Liang[‡], Yunxin Liu[‡], Enhong Chen[⋆]
[⋆]University of Science and Technology of China, China [‡]Microsoft Research, China

***Abstract*—Contrary to popular belief, mobile apps can spend a large fraction of time running "hidden" as background services. And, bugs in services can translate into crashes, energy depletion, device slow-down, etc. Unfortunately, without necessary testing tools, developers can only resort to telemetries from user devices in the wild. To this end, Snowdrop is a testing framework that systematically identifies and automates background services in Android apps. Snowdrop realizes a service-oriented approach that does not assume all inter-component communication messages are explicitly coded in the app bytecode. Furthermore, to improve the completeness of test inputs generated, Snowdrop infers field values by exploiting the similarity in how developers name variables. We evaluate Snowdrop by testing 848 commercially available mobile apps. Empirical results show that Snowdrop can achieve 20.91% more code path coverage than pathwise test input generators, and 64.11% more coverage than random test input generators.**

***Index Terms*—App background services, test input generation, Android Intents**

I. Introduction

While significant efforts have been invested in profiling the mobile app foreground activities [1], [2], [3], [4], we argue that proactively gaining visibility into app background services is equally important. Surprisingly, mobile apps can spend a significant fraction of time running hidden as background services [5]. Services can continue to run even after the associated app exits, e.g., pressing the home button on the mobile device or turning off the screen.

Services typically handle three types of operations on behalf of the app: (i) periodic app state refresh, (ii) server notifications, and (iii) long-running tasks that do not need user interactions (e.g., music playing and geo-location tracking). Surprisingly, Chen et al. [6] reported that apps can consume a significant amount of energy in the background – 45.9% of device battery drain are during the screen-off time, and 28.9% are due to apps with frequent background services. Rosen et al. [7] showed that ∼30.0% of network traffic are due to background services. Furthermore, background services can exhibit sensitive behaviors such as collecting location data [8].

In this work, *we argue that proactively testing background services can benefit from systematically automating through the service life cycle*. And, such testing can complement the foreground UI testing that many app developers are already doing. This is different from passive measures against misbehaving services – many user-centric tools focus on limiting background services at run-time [9], [10], [6], [11], and other approaches wait and collect telemetries from user devices in the wild [5]. These tools are passive measures against misbehaving services, and should not replace app testing.

Current app testing tools are mostly for UI testing, and they do not consider background service's unique execution model. First, services are "hidden" and not user-interactive, so there is no guarantee that they are reachable through any means of UI automation. Instead, developers typically rely on passing system-level inter-component communication (ICC) messages, or Intents, to start services. While our testing scenario seems applicable to related efforts in inferring Intent payloads [12], they strongly assume all Intents are explicitly coded by the developer. Furthermore, many efforts cannot reasonably infer valid values for arbitrary developer-specified fields in Intents [13], [14], [15], [16], [17].

To this end, we present *Snowdrop* – a developer-facing fuzzing framework that systematically discovers an app's hidden services and automates their lifecycle to cover all code paths. Conceptually, Snowdrop generates a more complete set of test inputs, by considering both trigger inputs and execution inputs. Taking advantage that most app packages are in bytecode that can be decompiled into intermediate representations, Snowdrop can perform static analysis without any developer overhead. Snowdrop addresses the following **challenges** in achieving the testing coverage.

First, to test all services of an app, Snowdrop realizes the service-oriented approach that generates test inputs by first localizing all services. This is different from the ICC-oriented approach [12] that simply identifies Intents coded in the app bytecode. In fact, if developers opt implicit Intents in their apps, the ICC-oriented approach can have a lower testing coverage as the target service of an implicit Intent is decided by the OS at run time. Furthermore, empirical results show that UI monkey testing has limitations in that ∼50.32% of services cannot be triggered by UI invocations.

Second, for the completeness of test inputs generated, we aim to infer developer-specified fields in Intents, e.g., `extras` key-value pairs. Since these fields can syntactically take arbitrary values, challenges arise from selecting a value that has the appropriate semantics from the vast solution space. While related efforts generate random test inputs or assume that all relevant value assignments are in the bytecode, Snowdrop exploits the similarity in how developers name variables. Our heuristic extends text classification with feature vector [18] from the natural language processing (NLP) community.

Third, to exercise all code paths of each identified service, Snowdrop generates execution inputs that include return values of system APIs. We argue that the low complexity of typical app services suggests that the proven pathwise test data generation [19] is feasible. Furthermore, since each hardware and system component can have a set of corresponding APIs, Snowdrop maintains finite-state models to ensure consistency across generated execution inputs.

We summarize our **contributions** as follows. This paper motivates the need to proactively test app background services, and lays the necessary foundation – a whitebox testing framework for background services. We address challenges in building a such framework, and implement an up-and-running system. Empirical results and case studies from 848 popular Android apps support Snowdrop's potential and practicality. Without human inputs, Snowdrop can achieve 20.91% more service code path coverage than existing IC3-based pathwise test input generators, and 64.11% more coverage than random test input generators. Finally, Snowdrop reported crashes on 12.05% of background services in our app pool, and we share our investigation findings in case studies.

II. Background

This section highlights hidden impacts of services on user experience, and describes unique properties of services that testing needs to consider.

A. Hidden Impacts of Services

Several measurement studies have tried to quantify previously unaware impacts of services on user experience.

Chen et al. [5] studied the telemetry of energy consumption of 800 Android apps on 1,520 Galaxy S3 and S4 devices. They found that 45.90% of device battery drain are during the screen-off time (i.e., only background services are running). For 22.5% of apps, the background energy expenditure accounts for more than 50% of the total energy.

Lee et al. [20] reported app processes can spend 16× more time in the background than being in the foreground. In fact, many apps periodically generate network usage in the background, regardless of user activities. For example, Facebook was observed to initiate network traffic every 20 minutes. Furthermore, Rosen et al. [7] looked at 342 apps, and noticed that up to 84% of the total network-related energy consumption happen in the background.

B. Android Background Services

Services are suitable for operations that do not need user interactions. Depending on the scenario, developers can choose one of the three interfaces – BroadcastReceiver handles short-running tasks (less than 10 seconds), Alarm implements time-based triggers for short operations, and Service is suitable for long-running non-interactive tasks. Since Service has a larger impact on the app performance and resource consumption, this paper mainly focuses on this type of background services.

Lifecycle. Since it is not easy to visually observe the lifecycle of services, their behavior might not be as expected. In Android, service can be further categorized into being unbounded and bounded. While both can be started at any time, unbounded service can actually continue running beyond the app's visually-perceivable lifecycle (rather than until the app exits). We elaborate on the lifecycle of unbounded services next.

TABLE I
Many types of service-triggering system events cannot be accomplished with only UI automation. These cases account for 50.32% of instances in our pool of 848 apps.

(a) Reachable through UI

Event trigger	**%**
Event callbacks	19.18
App launch	12.74
UI invocation	8.84
App pause	8.91

(b) Unreachable through UI

Event trigger	**%**
Broadcasts	25.69
ICC from services	13.43
App resume	7.57
Timer	3.63

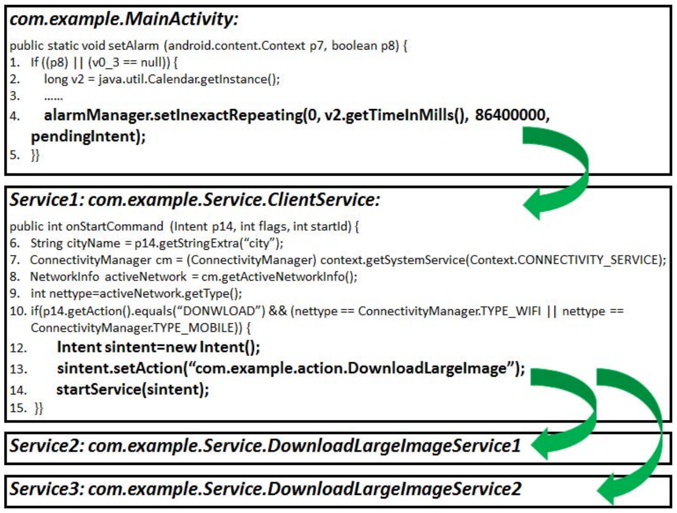
While the code to start services depends on the category of service used, the general idea is similar – inter-component communication (ICC), or Intents. For unbounded services, the Intent is sent through the `startService` method. As we describe in the next subsection, Intents can have arbitrary payloads as defined by the developer. Therefore, determining the proper payload for fuzzing is crucial. In addition to running to completion (e.g., `stopService` or `stopSelf` method), service can also be forced to terminate by the operating system's Low Memory Killer (LMK). The developer needs to properly handle the premature termination to avoid any state loss (e.g., `onDestroy` method).

ICC Messages (Intent). The Intent contains six fields specifying the delivery of an inter-component communication (ICC) message – ***(1)*** `component`: the target component name, which can be explicitly specified by a string of full-qualified class name, or can be an abstract action whose target component is implicitly decided by the OS at run time. Therefore, Intents can be explicit and implicit by distinguishing whether specifying the `component`. ***(2)*** `action`: a string to implicitly indicate the action to perform, ***(3)*** `uriData`: a Uri object expresses the data to operates on for the target `component` , ***(4)*** `datatype`: a string indicates the type of the `uriData`, ***(5)*** `category`: a string represents the additional information about the action to execute, ***(6)*** `extras`: consists of pairs of `extras.key` and `extras.value`, i.e., key-value pairs for storing any additional information. `extras.key` is of string type, and `extras.value` can take arbitrary type.

III. Gaps of Existing Testing Tools

A. Limitations of UI Monkeys

UI monkeys automate an app by invoking one UI element at each step. While UI event handlers can send Intents to start services, not all services are programmed to be triggered in this way. Fig. 1 shows an example of this case where `Service1` is indirectly triggered by a timer. To quantify this limitation of UI monkeys, we looked at our pool of 848 popular apps, and we counted the number of services whose `startService` methods are detected in UI handlers. Table I suggests that ∼50.32% of services are not reachable through UI invocation.



```
com.example.MainActivity:
public static void setAlarm (android.content.Context p7, boolean p8) {
If ((p8) || (v0_3 == null)) {
    long v2 = java.util.Calendar.getInstance();
    ......
    alarmManager.setInexactRepeating(0, v2.getTimeInMills(), 86400000,
  pendingIntent);
}}

Service1: com.example.Service.ClientService:
public int onStartCommand (Intent p14, int flags, int startId) {
String cityName = p14.getStringExtra("city");
ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
int nettype=activeNetwork.getType();
if(p14.getAction().equals("DONWLOAD") && (nettype == ConnectivityManager.TYPE_WIFI || nettype ==
  ConnectivityManager.TYPE_MOBILE)) {
      Intent sintent=new Intent();
      sintent.setAction("com.example.action.DownloadLargeImage");
      startService(sintent);
}}

Service2: com.example.Service.DownloadLargeImageService1

Service3: com.example.Service.DownloadLargeImageService2
```

Fig. 1. Not all Intents are explicitly coded. UI starts `Service1` with an alarm, and `Service2` is started by an implicit Intent. Being service-oriented, Snowdrop ensures the testing coverage without worrying any implicit redirection by the OS.

B. Limitations of Intent Inference Tools

As the ICC message is the direct way to start Android background services, ICC-oriented approaches try to localize all Intents explicitly coded in the app bytecode, and infer field values for these identified Intents based on some heuristics. As this subsection discusses, with respect to automating services, the testing coverage of these ICC-oriented approaches depends on two factors: ***(1)*** the number of Intents identified, and ***(2)*** the correctness of field values inferred, especially for developer-specified Intent fields (e.g., `extras`).

First, localizing all Intents explicitly coded in the app bytecode can have low testing coverage. As Fig. 1 illustrates, Android services can be triggered by implicit Intents [21]. And, implicit Intents introduce the uncertainty of which services are actually called by the OS at runtime. We quantify this coverage of automating services with IC3 [12]. Since all runnable background services need to be declared in the app manifest file, we use the manifest file as the ground truth (and remove dummy declarations not implemented). Empirical results show that Intents IC3 can find correspond to only 83.22% of services in our app pool of 848 popular apps.

Second, many Intent field inference heuristics do not consider the field semantics (e.g., URL, city name, geo-location coordinates). Since developer-specified fields such as `extras` can syntactically take arbitrary values, challenges arise from selecting the appropriate value from the vast candidate space. Some tools simply rely on developers to provide most test inputs [14]. To reduce the manual effort, *random test input generators* generate random Intent field values [13], [15], [16], [17]. While random generators have a low complexity, it does not guarantee test input quality.

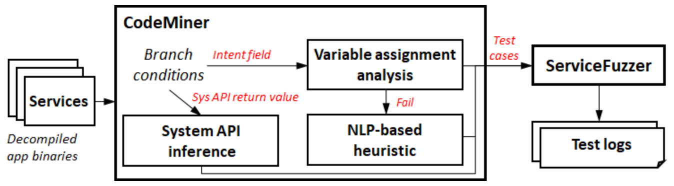


Fig. 2. Architectural overview of Snowdrop: (1) analyzing decompiled app bytecode to generate test inputs with CodeMiner, and (2) automating services with ServiceFuzzer.

IV. System Overview and Architecture

Snowdrop tackles the challenges of generating test input data necessary to exercise all code paths of every service in an app. This section discusses two major aspects in the design of Snowdrop: what test inputs are necessary, and how these test inputs can be automatically generated. These two questions impact the overall testing coverage.

Each test case (c.f. Def IV.1) maps to one code path of a service, and it contains *service trigger inputs* (c.f. Def IV.2) and *service execution inputs* (c.f. Def IV.3). The former is necessary to properly start a service, and it includes (*i*) service name and (*ii*) control dependencies among services. The latter dictates the control flow of service execution, and it includes (*i*) Intent fields (e.g., `extras`) and (*ii*) return values of system APIs called. Each code path of every service would have one set of trigger inputs and execution inputs.

Definition IV.1. A **test case** contains one instance of service trigger input and service execution input. Each case maps to one code path of a service.

Definition IV.2. We define a **service trigger input** as $\{c, \{cDen_1, ..., cDen_n\}\}$. c is the name of background service. $cDens$ are services that can send Intents to start this service.

Definition IV.3. We define a **service execution input** as $\{Intent, API_1, API_2, ...\}$. $Intent$ is $(p, p.v)$, where p and $p.v$ represent a pair of Intent field and value, respectively. Let API be $(a, a.v)$, where a and $a.v$ represent a pair of system API and return value, respectively.

Fig. 2 illustrates the system architecture. The developer submits compiled app packages. Since mobile app packages are typically in intermediate representations (e.g., Java bytecode), Snowdrop decompiles app bytecode to get readable bytecode, manifest and configuration files. Then, as this section elaborates next, the CodeMiner module uses static analysis to generate both trigger inputs and execution inputs. ServiceFuzzer takes the generated inputs to dynamically test individual background services.

CodeMiner. Unlike the ICC-oriented approach that assumes all Intents are explicitly coded in app bytecode, Snowdrop opts a *service-oriented approach* to generate test inputs – it localizes the code block of individual services, and then statically analyzes how these services access Intent fields.

CodeMiner takes three main considerations. First, there can be control dependencies among services, i.e., one service starts another service and may pass data as Intent values (c.f. §V). Second, generating execution inputs requires CodeMiner to look at each code control path in the service, and CodeMiner grounds on the proven pathwise approach [22] and constraint solving (c.f. §VI). Although the pathwise approach has been shown to scale poorly with respect to the code complexity, it is practical for testing app background services as they have a much lower complexity – an average of 8.6 code paths per service. Third, values of Intent fields can be arbitrary strings (e.g., `uriData` and `extras`), which complicates value inference without knowing the field semantics. For instance, the developer can store the video URL in `extras`. Therefore, CodeMiner adopts a heuristic that leverages the similarity in how developers name variables (c.f. §VI-A).

ServiceFuzzer. For individual instances of test inputs generated, ServiceFuzzer injects crafted explicit Intents (with off-the-shelf tools such as the Android Debugging Bridge), and it manipulates system API call return values (with off-the-shelf tools such as Android Xposed [23]). If the service dependency graph indicates that a service can be started by another service, then ServiceFuzzer does not try to directly automate it.

ServiceFuzzer takes two additional considerations (c.f. §VII). First, a testing session terminates when either the service finishes execution, or the developer-specified termination condition is met. Second, since Android does not provide performance counters at per-thread level, disaggregating counters is necessary to extract per-service information.

V. Service Trigger Inputs Generation

This section discusses techniques that CodeMiner uses to generate service trigger inputs.

A. Variable Assignment Analysis

Variable assignment analysis is an information flow analysis to keep track of how a value is passed from variables to variables. In the case of Snowdrop, it identifies variables in the code that originate from either Intent fields or system API return values. Specifically, given a variable in the code, CodeMiner starts backward analysis on each line of code from that point. And, it marks variables that have assignments leading to the target variable.

For Android apps, variable assignments can happen in four main places: ***(1)*** the variable is assigned somewhere at the caller method, and passed as an method argument, then CodeMiner would search all the call methods for the value. ***(2)*** the variable is assigned as a return value of a callee method, then CodeMiner would search the callee method for the returned value, ***(3)*** the variable is a class field variable, which is assigned somewhere within the class scope. CodeMiner analyzes the DFG (Data-Flow Graph) to search the field for the last time assigned, ***(4)*** the variable is assigned within the method scope.

B. Service Dependency Graph Construction

In the service dependency graph, each node represents a service, and directed edges indicate the caller-callee relationship

TABLE II
Percentage of Service Dependencies that can successfully start background services.

Service caller	Num services (%)	Success rate (%)
App on-launch	15.52	94.88
BroadcastReceiver	31.51	92.72
Service	16.41	90.06

(i.e., the source node can send Intent to start the destination node). If a service can be started by another, Snowdrop can save time by not directly automating it. We note that false negatives in capturing the dependency do not impact the testing coverage, as Snowdrop will automate any service with no known dependency.

Discover Background Services (c)**.** The number of background services that CodeMiner can find for subsequent test input generation impacts the overall testing coverage. Since all runnable background services need to be declared in the app manifest file, CodeMiner parses the Android manifest file to list all background services of an app. However, it is possible that the developer declares services that are never implemented. To filter out such cases, CodeMiner then searches each declared service in the decompiled intermediate representations. Specifically, each Android service should extend the `Service` class.

Locate Caller Component ($cDen$)**.** Given a target service name, we now identify its caller service. By definition, the caller service would have code to construct the corresponding Intent. Therefore, for each Intent in the code, we perform variable assignment analysis to find the target component name. If the name matches our target service, we look at the component that sends this Intent. If the component is another service (e.g. Service or BroadcastReceiver), then we add a directed edge in the service dependency graph. We also note Intents that reside in `onCreate` of the main activity, which means our target service would be automatically started after the app launch.

C. Microbenchmarks

This subsection characterizes service dependency graphs in our pool of 848 apps (c.f. §VIII-A). Table II shows the distribution of the three components that can send Intents to start services indirectly: app on-launch, Service, and BroadcastReceiver. Interestingly, there are 16.41% of services that have a dependency with another service, and 90.06% can be started successfully. In other words, there is no need for Snowdrop to start services that can be indirectly started. Table II suggests that a benefit of leveraging service dependency graph is the reduction in the number of test cases by 58.72%.

VI. Service Execution Inputs Generation

This section discusses techniques that CodeMiner uses to generate service execution inputs.

CodeMiner follows the standard practice of representing a code path by its constraints that need to be solved. For

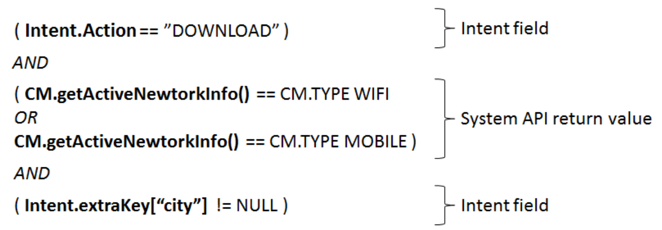

```
( Intent.Action == "DOWNLOAD" )                        } Intent field
AND
( CM.getActiveNewtorkInfo() == CM.TYPE WIFI
OR                                                     } System API return value
CM.getActiveNewtorkInfo() == CM.TYPE MOBILE )
AND
( Intent.extraKey["city"] != NULL )                    } Intent field
```

Fig. 3. Pathwise constraints of a code path for `Service1`. Satisfying constraints requires fuzzing Intent fields and system API return values.

instance, Fig. 3 shows one set of pathwise constraints for `Service1` in Fig. 1. Constraints on a path are first solved individually, to obtain a set of possible values. Then, for constraints that have several sets (e.g., network info in our example), sets are merged by the operation of union or intersection, according to the logical operators in the path.

The rest of this section discusses how individual constraints are solved, and this requires inferring both Intent field values and system API return values.

A. Intent Field Inference

As mentioned before, an Android Intent object has six fields: `component`, `action`, `category`, `datatype`, `extras` (<`extras.key`, `extras.value`>), `uriData`. This component outputs a set of (p, $p.v$). We note that `extras.key` and `extras.value` make up a key-value pair. Thus, when $p \notin$ {`action`, `component`, `uriData`, `category`, `datatype`}, p represents the `extras.key`. For example, in the set {(`action`, `android.intent.action.View`), (`username`, `david`)}, `username` represents the `extras.key`, and `david` is the corresponding `extras.value`.

Since Android confines what values the first four fields can take, developers typically hardcode their values, which makes it easy to run variable assignment analysis (c.f. §V-A). Both `uriData` and `extras.value` are more challenging to infer than other fields, because they can syntactically take on arbitrary values. While a naive way is to randomly generate data for these two fields, CodeMiner improves the semantical correctness of value inference with NLP-based heuristic. Next, we elaborate on this process.

Inferring `uriData` Field. `uriData` stores URI address. While some developers hardcode the URI string, `uriData` can also be dynamically assigned at runtime – as we discuss later, table VI suggests only 0.63% `uriData` can be inferred from searching for hardcoded strings. Since `datatype` specifies `uriData`'s MIME type, we can assign a URI that matches the MIME type. If the `datatype` field is not specified in the Intent, we can use another Intent field, `action`, to infer the MIME type. This is because the MIME type can have a connection with the Intent's `action`. For instance, if `action` is `ACTION_EDIT`, `uriData` most likely contains the URI of the document to edit.

Then, we classify URI MIME types into the following commonly used groups: *text (plain)*, *image*, *application*, *multipart*, *video*, *x-token*, *audio*, *message*. These are realized by matching `datatype` and `action`. Each group contains a list of candidate URIs for the address of objects pre-loaded on the emulator, or for popular webpages.

Inferring `extras` Field. `extras.value` can take on arbitrary types and values, which depend on the usage scenario ranging from storing web URLs to city names. From analyzing our app pool, we made an observation that developers typically use similar key names for the same purpose. An example is "music_url" and "songURL", which both reference to some web URLs of audio files. We note that, in this example, while it is practically infeasible to guess the exact URL string, providing a compatible string is sufficient to carry on the service execution. Therefore, inferring the semantics of `extras.value` is crucial for inferring the value.

Building on efforts from the natural language community, CodeMiner combines the Word2Vec [18] tool and the Support Vector Machine (SVM) [24] classification algorithm. Given a word, Word2Vec computes the feature vector for it. Snowdrop uses a popular Word2Vec model [25] already trained to recognize word semantics. So, words that are semantically similar should have similar feature vectors. We note that feature vectors are in mathematical representation, and they can be clustered by data classification algorithms such as SVM.

To train SVM to cluster similar feature vectors, we first get vectors of 5,724 `extras.keys` in our app pool. For key name composing of multiple words, we use the standard practice of averaging over individual vectors. Then, we label each `extras.key`'s vectors with 24 groups: *name*, *id*, *text*, *title*, *number*, *action*, *video*, *type*, *folder*, *user*, *audio*, *file*, *version*, *message*, *error*, *date*, *time*, *alarm*, *url*, *image*, *location*, *password*, *json*, *widget*. These 24 groups are the top groups after we apply k-means clustering [24] on all `extras.keys`.

Each group has a set of candidate values, such as a string that represents the current city name. Candidate sets are mainly from four sources: ***(1)*** app-based data include widget ID, package name, local storage path, etc, ***(2)*** environment-based data include current date time with different format strings, ***(3)*** file-based data cover specifications and formats, e.g., the format and size of pictures, and ***(4)*** URL-based data include URL addresses for web pages, images, audio, videos, etc.

With the trained Word2Vec model and the SVM model, CodeMiner first classifies each new `extras.key` and then assigns a candidate value to the corresponding `extras.value` field.

B. System API Return Value Inference

System APIs belong to the `Android` namespace, and their usage has an "android." or "java." prefix. This observation simplifies the task of determining which variables hold values derived from some system API return values. Resource-related system APIs considered are listed in Table III.

However, since a group of system APIs can change one mobile device component (e.g., GPS), the challenge lies in generating test data that are consistent to the hardware state. To this end, CodeMiner maintains a finite state machine (FSM)

TABLE III
POPULAR ANDROID SYSTEM APIS CALLED BY BACKGROUND SERVICES.

System API	%	System API	%
log	83.46	file_info	3.23
network_info	31.05	account_info	2.03
wakelock	17.38	audio	1.94
calendar_info	16.00	bluetooth_info	1.29
database_info	15.53	system_settings	0.65
file	14.04	sms_mms	0.37
network	13.22	account_settings	0.09
location_info	11.65	synchronization_data	0.09
unique_identifier	7.67	contact_info	0.09

for typical hardware components on mobile devices. Fortunately, most mobile devices exhibit a similar set of hardware capabilities: geo-localization, data I/O, motion-related sensors, etc. In addition, Android provides standard abstractions to hide individual hardware components' intrinsic properties. For example, sensors basically have three reachable states: *on*, *off*, *sampling*. For each system API that CodeMiner needs to generate test data, CodeMiner keeps track of the current state of hardware components. In the sensor example above, if the sensor has been activated by registering with `SensorManager`, then its status should be *on*.

C. Microbenchmarks

We evaluate the use of the SVM-based classification algorithm, by comparing with the human baseline. Specifically, with 5,724 `extras.key` strings from our app pool, we manually label them into 24 groups (c.f. §VI-A). Then, we randomly allocate 60%, 20% and 20% of these strings as the training set, validation set, and testing set, respectively. We use cross-validation to choose the parameter c=10 which trades off misclassification against simplicity of the decision surface from the set: 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 50, 100. The accuracy measured by the testing set is 79.55% over the 24 groups (as compared to the random guess of an accuracy of ∼4.2%), with a recall of 78.41%, and a precision of 80.81%.

VII. ADDITIONAL PRACTICAL CONSIDERATIONS

This section discusses practical considerations for ServiceFuzzer to test individual background services, with test inputs generated by CodeMiner.

A. Automating Background Services

Before injecting Intents to start services, ServiceFuzzer launches the app to allow the opportunity to properly initialize. For example, many apps check for environment dependencies, e.g., the database tables. Then, ServiceFuzzer pauses the app foreground activity, by simulating the user pressing the device HOME button.

ServiceFuzzer then selects nodes without any incoming edges in the service dependency graph, as these nodes represent services (c) without any dependency ($cDen$s). For each of these services, ServiceFuzzer turns corresponding sets of test data into individual explicit Intents. Specifically, c (in the service trigger inputs) becomes the Intent's target `component`, and pairs of p and $p.v$ (in service execution inputs) become Intent fields.

The execution of service code paths is guided by both Intent fields and any system API return values. The former is in the Intent crafted in the previous step. The latter requires ServiceFuzzer to intercept system API calls by the service, and this can be done with off-the-shelf tools such as Xposed. For each system API call intercepted, ServiceFuzzer searches in the execution inputs for a pair of a and $a.v$ that matches the system API. The return value is then manipulated to be $a.v$. For example, we can intercept the return value of `activeNetwork.getType()` with an integer `ConnectivityManager.TYPE_WIFI` or `ConnectivityManager.TYPE_MOBILE` in Fig. 1.

B. Terminating a Testing Session

To simulate different termination conditions, Snowdrop allows developers to select from the following two termination strategies for each testing session.

Graceful Termination. This termination strategy allows the service to run to completion. ServiceFuzzer passively infers the lifecycle by intercepting `onStartCommand`, `onBind`, and `onDestroy` callback events.

Forced Termination. This termination strategy kills the service when the developer-specified timer fires. In addition, the developer can instrument calls to our debugging API, `SNOWDROP_KILL()`, in the app bytecode. Upon intercepting this API call at run time, ServiceFuzzer would kill the app main process and spawned processes, to simulate the termination forced by the OS.

C. Disaggregating Logs

Snowdrop provides resource utilization logs including CPU and memory. Providing *per-service* logs can be challenging, as the Android platform provides performance counters at app-level, not component-level. Snowdrop exercises log disaggregation as below.

We record the service's start and end time, and attribute counters in this lifetime period to it accordingly. The service lifetime typically can be identified with the three callbacks: `onStartCommand`, `onBind`, `onDestroy`. We note that there are cases of concurrent threads: a service is created as a thread within the app main process, or even multiple services. To handle these cases, we attribute the resource utilization equally, and this is a standard technique used by app data analysis tools [26].

VIII. EVALUATION

This section is organized by the following major results: ***(1)*** Without human inputs, Snowdrop can achieve 20.91% more service code path coverage than existing pathwise test input generators, and 64.11% more coverage than random test input generators. ***(2)*** 92.17% of Intents generated by CodeMiner are not malformed or invalid – more than 21.19% higher than comparison baselines. ***(3)*** Test input generation takes an average of 20.06 sec for one Android service. This makes Snowdrop feasible for testing background services in practice.

A. Methodology and Implementation

We implemented Snowdrop in Python and Java – CodeMiner has 17,916 lines of code (LOC), ServiceFuzzer has 7,814 LOC, and logging modules have 4,463 LOC. CodeMiner disassembles and decompiles Android app bytecode with Androguard [27]. ServiceFuzzer runs inside Genymotion [28], one of the fastest emulators available. ServiceFuzzer exercises the standard practice of injecting Intents and killing services, with Android Debugging Bridge (ADB). And, it uses Xposed [23] to manipulate system API return values. To calculate code path coverage, we modify BBoxTester [29] with the popular tool JaCoCo [30] to first list all code paths in services. Then, we cross-reference this full list with our own list of executed branches, to calculate the number of exercised paths.

Evaluation Metrics and Comparison Baselines. To evaluate the effectiveness, we adopt the metric of path coverage, or the number of branched control paths in a service that a testing tool executes. We also use the test input generation time as another evaluation metric.

With these metrics, evaluations are based on the following comparison baselines. First, for representing random test input generators, one baseline is Intent Fuzzer [15], an ICC-based fuzzing for Android apps. It relies on static analysis to identify all Intents and their mandatory fields, and then randomly generates field values. Another baseline is NullIntentFuzzer [14], a fuzzing tool that focuses on crafting Intents with only the service `component` name. Second, as there is no off-the-shelf pathwise test input generators for app services, we implemented one with IC3 [31]. IC3 is the current state-of-the-art solution for Intent fields inference.

Methodology. Our app pool consists of 848 popular apps on the Google Play store. To minimize any test bias towards certain app categories, we take top 53 apps from each of the 16 popular app categories (i.e., News, Music, and Social). Collectively, these 848 apps have a total of 1,968 Services.

To run experiments, we generate test inputs for all apps, with Snowdrop and comparison baselines. For Snowdrop and IC3, each code path in identified services has a test case, and we let a test run for five minutes after injecting the Intent via ADB. Since it is difficult to force random test input generators to target a particular code path, we let Intent Fuzzer and NullIntentFuzzer run as many rounds as possible within the same time window, for fairness.

Verifiability. Factors that might influence the reproducibility of results include ***(1)*** the version of apps in our pool, ***(2)*** the version of open-sourced tools that Snowdrop uses, and ***(3)*** the hardware spec of our testing environment. For the first two factors, we have archived the downloaded packages. For the last factor, we are running Windows 10 on Intel i7-3770 CPU and 16 GB of RAM.

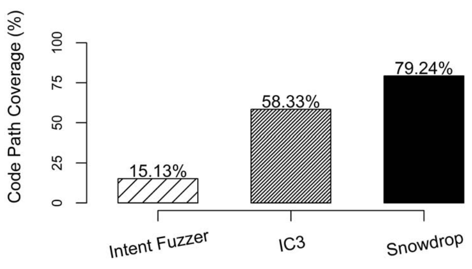


Fig. 4. Service code path coverage for different test input generators. Each tool tries to generate test inputs to execute as many code paths in services of our app pool as possible. By inferring all Intent fields and system API return values, Snowdrop can achieve better coverage than IC3 and Intent Fuzzer.

B. What is the Service Code Path Coverage Achieved?

We start by empirically measuring and comparing Snowdrop's service code path coverage. As mentioned above, the ground truth is computed by BBoxTester [29] and JaCoCo [30]. We generate the test input for each code path, with Snowdrop and two other baselines, Intent Fuzzer and IC3. These two baselines provide visibility into existing random and pathwise test input generators. A test case represents one code path, and it successfully completes if all branches on the code path are executed. As Fig. 4 shows, excluding services in third-party libraries, Snowdrop successfully executes 79.24% of all 16,925 service code paths in our app pool. This is 20.91% more coverage than IC3 and 64.11% more coverage than Intent Fuzzer.

There are two cases where a test case can not completely execute a code path: ***(1)*** a branch cannot be satisfied, and ***(2)*** execution exceptions due to malformed or invalid Intents. The following two subsections delve into these two cases.

C. How Well Do Test Inputs Satisfy Branch Conditions?

One factor affecting the service code path coverage is whether the test input can satisfy all branches on its intended code path. There are two main reasons behind unsatisfiable branching conditions: ***(1)*** branching conditions are hard-coded to be false, or ***(2)*** branching arguments cannot be externally fuzzed. To quantify these reasons, we classify branching arguments by their types: *Constant*, *Non_Sys_API* (e.g., developers' own functions), *Sys_API*, and *Intent_Field*. The last two classes can be externally fuzzed. While ServiceFuzzer fuzzes *Sys_API* return values through interception (c.f. §VIII-A), we do not intercept *Non_Sys_APIs* as they also contain developer code that needs to be tested.

Table IV shows the distribution of possible variations of branching conditions. Each variation has one of the two tags: *Fuzzable* (i.e, the branching condition arguments can be fuzzed to satisfy) and *Unfuzzable* (i.e, otherwise). Given that only *Sys_API*, and *Intent_Field* can be external fuzzed, there are some branching statements out of scope for Snowdrop.

Table V shows that Snowdrop can successfully fuzz 89.41% of branching conditions. We note that cases involving *Sys_API* tend to have a relatively lower percentage. And, this is a known problem in the community [32], as certain data types of return

TABLE IV
DISTRIBUTION OF DIFFERENT BRANCHING CONDITIONS BY THE TYPE OF THEIR ARGUMENTS. GIVEN THAT ONLY *Sys_API*, AND *Intent_Field* CAN BE EXTERNALLY FUZZED, THERE ARE SOME BRANCHING STATEMENTS OUT OF SCOPE FOR SNOWDROP.

Tag	Argument 1	Argument 2	%
Unfuzzable	Constant	Constant	8.83
Fuzzable	Sys_API	Constant	10.71
Fuzzable	Sys_API	Sys_API	0.29
Fuzzable	Sys_API	Intent_Field	1.55
Fuzzable	Intent_Field	Intent_Field	0.03
Fuzzable	Intent_Field	Constant	52.83
Unfuzzable	Sys_API	Non_Sys_API	0.38
Unfuzzable	Non_Sys_API	Intent_Field	2.56
Unfuzzable	Non_Sys_API	Constant	22.06
Unfuzzable	Non_Sys_API	Non_Sys_API	0.78

TABLE V
NUMBER OF FUZZABLE BRANCHING CONDITIONS THAT SNOWDROP CAN SATISFY.

Argument 1	Argument 2	Satisfiable (%)
Sys_API	Constant	84.35
Sys_API	Sys_API	72.71
Sys_API	Intent_Field	92.71
Intent_Field	Intent_Field	100.0
Intent_Field	Constant	97.28

values are difficult to fuzz, which include arrays, iterators, the pair in a map, a set, I/O operations, content providers, or even some complex data objects (e.g., bitmap).

Finally, we note the use of *Non_Sys_API*s varies among app categories. Fig. 5 organizes Snowdrop's code coverage by the app category. Interestingly, it suggests that Music and Photography apps generally achieve lower coverage. Photography apps tend to use specialized library functions such as calculating the rescaling size for bitmap files. Similarly, Music apps contain many developer-defined objects such as a singer profile object that is generally difficult to solve for test input generators.

D. What is the Correctness of Intent Inference Heuristics?

Another factor affecting the service code path coverage is the correctness of Intents generated. Specifically, malformed and invalid Intents can cause run-time exceptions, and these exceptions force tests to terminate unexpectedly. For instance, a tool can assign arbitrary string values to `extras` key-value pairs that expect well-formed strings such as web URLs or media file paths. In other cases, numeric values can have a meaning such as GPS longitude and latitude. Fig. 6 shows that 92.17% of Intents generated by Snowdrop can successfully run to the test completion. Interestingly, this is 21.19% and 48.35% more successful cases than IC3 and Intent Fuzzer, respectively. Since IC3 and Intent Fuzzer do not consider the field semantics, most exceptions caught (e.g., `NumberFormatException` and and `NullPointerException`) are due to malformed and invalid test inputs.

Next, we discuss malformed Intents generated by CodeMiner. First, 51.33% of these cases belong to services in

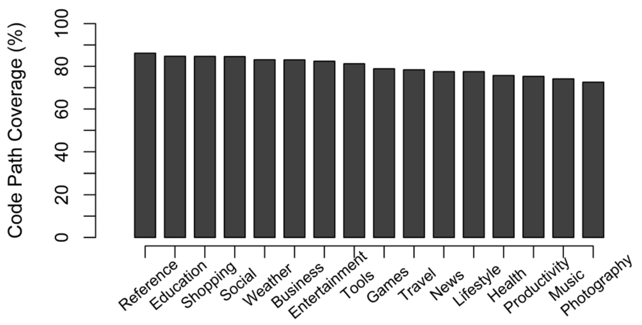


Fig. 5. Depending on the type of branching conditions and data structures in the code, some app categories are more difficult to generate test inputs for.

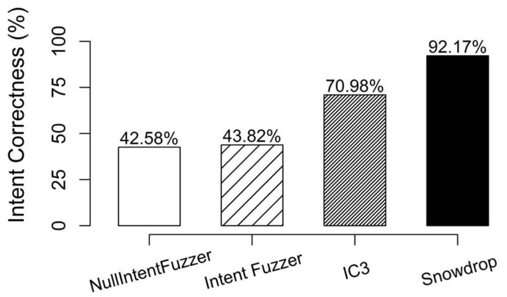


Fig. 6. Malformed or invalid Intents can cause run-time exceptions, and force a test to terminate unexpectedly. Of all Intents generated, Snowdrop generates the highest percentage of correct Intents. For comparison baselines, many exceptions include `NumberFormatException` and `NullPointerException`.

third-party libraries, e.g., advertisement and app analytics tracking. One observation is that these third-party libraries have Intent fields with uncommon naming convention, e.g., EXTRA_SITUATION and FRONTIA_USER_AIPKEY. Second, 12.67% of unsuccessful cases are due to widget update services, which require the right widget ID. Third, 4.67% of unsuccessful cases are services for licensing or signing information, which require the right licensing information. While being out of scope for this paper, it is possible to implement a human feedback loop to address these limitations.

Finally, we benchmark the two inference heuristics of CodeMiner to understand how they contribute to the correctness of Intents. As we mentioned before, CodeMiner first tries variable assignment analysis (VAA), to find field value assignments in app binaries. Since developer-specified fields can be difficult to infer, CodeMiner then infers their values with NLP-based heuristic (NLPH). Table VI shows the effectiveness of the two strategies: variable assignment analysis (VAA) and NLP-based heuristic (NLPH). First, we observe that variable assignment analysis can handle most cases of `component`, `action`, `datatype`, `extras.key`, and `category`. The reason is that these fields have a predefined set of values, and most apps explicitly assign their values. Second, Table VI suggests that the situation is different for `uriData` and `extras.value`, where the developer can assign arbitrary values. In these cases, the inference relies

TABLE VI
PERCENTAGE OF INTENT FIELDS WHOSE VALUES CAN BE INFERRED BY VARIABLE ASSIGNMENT ANALYSIS (VAA) AND NLP-BASED HEURISTIC (NLPH). RESULTS ARE GROUPED BY THE INTENT FIELD TYPE.

Intent field	**Inferred by VAA (%)**	**Inferred by NLPH (%)**	**Unresolved (%)**
component	90.75	N/A	9.25
action	90.77	N/A	9.23
uriData	0.63	99.37	N/A
datatype	79.35	N/A	20.65
extras.key	96.63	N/A	3.37
extras.value	1.85	98.15	N/A
category	97.08	N/A	2.82

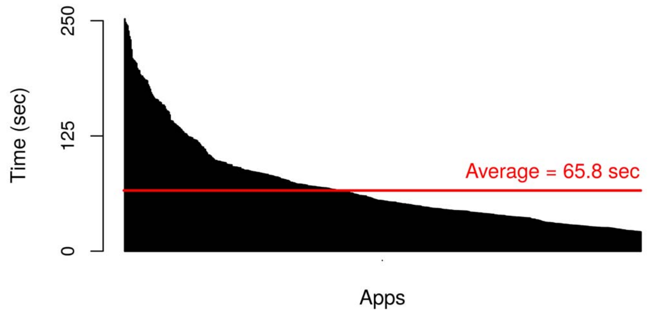


Fig. 7. Time taken to generate test inputs for each of our 848 apps. The computation cost of Snowdrop is practical for real-world usage.

on applying classification techniques to the variable name. Empirical results show that 99.37% of `uriData` and 98.15% of `extras.value` can be inferred this way.

Table VI also suggests that there are instances CodeMiner fails to infer. One reason is that CodeMiner ignores data types that cannot be analyzed by static analysis or sent over ADB: Parcelable, Serializable, and Bundle. In our app pool, while most `extras.values` are String and Integer, 12.16% of them are of these unsupported types.

E. Is the Computation Cost Feasible in Practice?

Measurements show that Snowdrop has a reasonable cost for real-world usage. Fig. 7 shows the average time to generate test cases for an app is 65.80 sec, which translates to 20.06 sec per Android app service. While being out of scope for this paper, optimizations on constraint solving will reduce this cost. For reference, we note that IC3 takes an average of 23.41 sec to generate the ICC message per service.

IX. CASE STUDIES

With Snowdrop, we tested our pool of 848 apps and found that 12.05% of background services have bugs. For example, 8.88% of apps have problems of poor exception handling. For analysis, we logged ADB runtime exceptions and various performance counters including CPU utilization, network tx/rx, power consumption, etc. We confirmed Snowdrop's findings by manually examining decompiled the bytecode of reported apps. This section highlights real-world bugs that Snowdrop reported on apps released in the wild.

A. Crashes and Poor Exception Handling

TED. Snowdrop detected *java.lang.NullPointerException: PowerManager$WakeLock.release() on a null object reference*. By manually examining the decompiled app bytecode, we found that the `com.pushio.manager.PushIOEngagementService` service does not initialize a WakeLock object before acquiring and releasing the lock in some code paths.

Huffington Post. Snowdrop detected a crash due to *java.lang.RuntimeException: WakeLock under-locked com.commonsware.cwac.wakeful.WakefulIntentService* in `downloadall.AutoOfflineService`. This error suggests that the WakeLock was released more times than it was acquired. We confirmed by decompiling the Huffington Post app. And, we found that the app does not check whether the WakeLock is acquired in `wakeful.WakefulIntentService`, which is called by `downloadall.AutoOfflineService`.

Sophos Mobile Control. Snowdrop detected *java.lang.NoClassDefFoundError: com.sonymobile.enterprise.Configuration* in `InstallCertificateService`. We manually inspected the decompiled bytecode – the `InstallCertificateService` tried to instantiate the *com.sonymobile.enterprise.Configuration* class, which does not exist in the expected third-party library, *sonymobile.enterprise*. One possible explanation is that the developer might have mistaken the library version.

B. Resource Usage Bugs

FOX TV Turkiye. Logs indicate that this app has many periods of extremely low CPU utilization. This observation suggests that the app might have prevented the CPU from being suspended to reduce energy consumption. We manually examined the decompiled app bytecode, and we found that the `PushAdService` was trying to hold *pushad.wakelock* until all async jobs completed. However, this `WakeLock` was not properly released in two of the code paths.

AirPlay/DLNA Receiver. AirPlay/DLNA Receiver wirelessly receives videos, pictures and music from nearby devices. Without any active streaming sessions, we noticed that the app had an unexpectedly high network traffic (~1 MB/s) when `WaxPlayService` was running. One potential impact of this behavior is the high energy consumption. We used WireShark [33] to investigate possible causes with network-level traffic traces, and we found that the app constantly sent out 599-byte MDNS/Bonjour requests. In addition, the app periodically sends a group of 60-byte TCP packets to the network gateway.

X. RELATED WORK

A. Mobile App Testing Frameworks

Most existing testing frameworks rely on UI automation, for workloads of touch-screen interactions and gestures. Appium [34] and UiAutomator [35] are popular frameworks with API to build UI-driven tests. PUMA [36] explores a flexible programming paradigm to UI-driven app analysis, and RERAN [37] addresses touch-sensitive record-and-replay. Testflight [38] is a framework for iOS developers to invite real-world users to exercise the app. AMC [39] reduces the amount of work that a human expert must perform to evaluate the conformance of vehicular apps. SmartDroid [40] identifies UI-based trigger conditions in Android apps to help privacy experts to quickly find the sensitive behaviors. RainDrops [41] envisions a split-execution model for building automated and contextual testing services to bring real-world contexts into mobile app testing.

Unfortunately, background services are hidden from users, and empirical data show $\sim$50.32% of services are not reachable through UI invocations. This observation motivates the need of Snowdrop.

B. Intent Inference Tools

Many testing scenarios rely on the ICC-oriented approach, or the ability to extract ICC messages coded in the app bytecode. For instance, ComDroid [42] and IccTA [43] analyze ICC messages to detect privacy leaks.

As §III-B discusses, there are also generic tools to facilitate the ICC-oriented approach. The first category of tools is simply a fuzzer that assumes most of the Intent structure would be given by the user [14]. These tools can impose a significant burden due to manually creating test inputs to achieve full coverage. To this end, random test input generators can generate random Intent field values [13], [15]. However, random data generators do not guarantee test input quality, nor the time to achieve full testing coverage [44]. Finally, some tools exploit app code knowledge to make informed decisions on Intent inference and generation. IntentFuzzer [16] intercepts API calls to learn keys in `extras`, and then randomly generates value for each key. IC3 [12] improves on Epicc [45], and it analyzes the code to infer constraints given by a code path on values that each field can take on.

While these ICC-oriented tools try to identify as many Intents as possible in the app bytecode, implicit Intents can be challenging for static analysis tools. Furthermore, since some Intent fields can syntactically take on arbitrary values, Snowdrop improves the inference correctness with NLP-based heuristic.

C. Visibility Into Background Services

The research community has recently started to explore problems related to background services. Chen et al. [5], [6] conducts a large-scale study, and telemetries from user devices reveal services can consume unexpected amount of energy. Then, they propose HUSH as a way to dynamically suppress background services on the device. TAMER [11] is an OS mechanism that monitors events and rate-limits the wake-up of app services on the device.

There are related efforts on code analysis. Entrack [26] aims to finely trace energy consumption of system services, and addresses challenges in accurate per-thread accounting. Pathak et al. [46] proposes a compile-time solution based on the classic reaching data-flow analysis problem to automatically infer the possibility of a no-sleep bug in a given app.

Snowdrop can complement these efforts, and it fills the gap of enabling developers to systematically test all services before the app release.

XI. Discussion and Future Work

System Limitations. As with any system, Snowdrop has limitations. First, dynamically loaded code can be challenging for Snowdrop to analyze, as the code might not be available until run time. Fortunately, most background services are not implemented in this fashion. Second, while the NLP-based heuristic improves the correctness of Intent value inference for Snowdrop, there are still cases where developers use meaningless variable names that complicate semantic understanding. As a future work, we are exploring additional means of inferring the semantics of variables.

Overhead of Pathwise Test Input Generation. The community has raised the concern that pathwise test input generation does not scale well with the software complexity. Fortunately, mobile app services generally have relatively low complexity, and empirical results suggest an average of 8.60 code paths per service.

Complementing UI Automation Testing. Many tools are available for developers to automate UI testing. We believe that ensuring app experience can benefit from testing both UI and background services. Since both aspects do not conflict with each other, Snowdrop can complement existing UI automation testing.

Applicability to Other Mobile Platforms. Background services that consume too much resources are also present in iOS and Windows apps. While concepts of Snowdrop are applicable to other mobile platforms, our current investigation suggests that there does not seem to be an easy way to trigger background services to run. We are exploring workarounds.

XII. Conclusion

Snowdrop fills the gap towards systematically testing app background services. It automatically generates test inputs to maximize service code path coverage. Empirical results show that Snowdrop can achieve higher coverage over existing random and pathwise test input generators. Furthermore, Snowdrop has uncovered service problems ranging from crashes to energy bugs, and it can well complement the existing UI automation based testing.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. This research was partially supported by grants from the Nature Science Foundation of China (Grant No. U1605251), and the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61325010).

References

[1] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing," in *The 20th Annual International Conference on Mobile Computing and Networking Paris, MobiCom*. ACM, 2014, pp. 519–530.

[2] B. Liu, S. Nath, R. Govindan, and J. Liu, "DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps," in *11th USENIX Symposium on Networked Systems Design and Implementation, NSDI*. USENIX, 2014, pp. 57–70.

[3] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and Scalable Fault Detection for Mobile Applications," in *MobiSys*. ACM, 2014, pp. 190–203.

[4] L. L. Zhang, C.-J. M. Liang, Z. L. Li, Y. Liu, F. Zhao, and E. Chen, "Characterizing privacy risks of mobile apps with sensitivity analysis," in *IEEE Transactions on Mobile Computing (TMC)*. IEEE, 2017.

[5] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone Energy Drain in the Wild: Analysis and Implications," in *SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems,SIGMETRICS*. ACM, 2015, pp. 151–164.

[6] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization," in *The 21th Annual International Conference on Mobile Computing and Networking Paris, MobiCom*. ACM, 2015, pp. 40–52.

[7] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen, "Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild," in *Proceedings of the 2015 Internet Measurement Conference, IMC*. ACM, 2015, pp. 339–345.

[8] S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users," in *Third ACM Conference on Data and Application Security and Privacy, CODASPY*. ACM, 2013, pp. 221–232.

[9] Android, "Android Restrict Background Data Usage," https://support.google.com/nexus/answer/2819524?hl=en, 2016.

[10] Apple, "iOS Background App Refresh ," https://support.apple.com/en-us/HT202070, 2016.

[11] M. Martins, J. Cappos, and R. Fonseca, "Selectively Taming Background Android Apps to Improve Battery Lifetime," in *USENIX Annual Technical Conference, USENIX ATC*. USENIX, 2015, pp. 563–575.

[12] SIIS Lab at Penn State Unverisity, "IC3: Inter-Component Communication Analysis for Android," http://siis.cse.psu.edu/ic3/index.html#banner, 2015.

[13] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2012, pp. 1–12.

[14] NCC Group, "Intent fuzzer," https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer/, 2012.

[15] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in *WODA+PERTEA*, 2014, pp. 1–5.

[16] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting Capability Leaks of Android Applications," in *ACM Asia Conference on Computer and Communications Security, ASIA CCS*, 2014, pp. 531–536.

[17] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *Proceedings of International Conference on Advance in Mobile Computing and Multimedia, MoMM*, 2013, pp. 68:68–68:74.

[18] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," in *Neural Information Processing Systems, NIPS*. NIPS, 2013, pp. 3111–3119.

[19] R. DeMillo and A. Offutt, "Constraint-based Automatic Test Data Generation," in *IEEE Transactions on Software Engineering, TSE*, vol. 17, no. 9. IEEE, 1991, pp. 900–910.

[20] J. Lee, K. Lee, E. Jeong, J. Jo, and N. B. Shroff, "Context-aware Application Scheduling in Mobile Systems: What Will Users Do and Not Do Next?" in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, Ubicomp*, 2016, pp. 1235–1246.

[21] Android, "Android Services," https://developer.android.com/guide/components/services.html, 2017.

[22] L. Clarke, "A system to generate test data and symbolically execute programs," in *IEEE Transactions on Software Engineering*, vol. 2. IEEE, 1976, pp. 215–222.

[23] rovo89, Tungstwenty, "Xposed," http://repo.xposed.info/module/de.robv.android.xposed.installer, 2016.

[24] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An Efficient k-Means Clustering Algorithm: Analysis and Implementation ," in *Transactions on Pattern Analysis and Machine Intelligence*, vol. 24. IEEE, 2002, pp. 881–892.

[25] "Word2Vec Toolkit," https://code.google.com/archive/p/word2vec/, 2013.

[26] M. Martins, J. Cappos, and R. Fonseca, "Entrack: A System Facility for Analyzing Energy Consumption of Android System Services," in *ACM International Joint Conference On Pervasive And Ubiquitous Computing, Ubicomp*. ACM, 2015, pp. 191–202.

[27] A. Team, "Androguard," https://github.com/androguard/androguard, 2015.

[28] Genymobile Inc., "Genymotion," http://www.genymotion.com/, 2016.

[29] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *2015 Tenth International Conference on Availability, Reliability and Security (ARES)*, 2015, pp. 501–510.

[30] JaCoCo Developers, "Jacoco java code coverage library," http://www.jacoco.org/jacoco//, 2016.

[31] Damien Octeau and Daniel Luchaup and Matthew Dering and Somesh Jha and Patrick McDaniel, "IC3," http://siis.cse.psu.edu/ic3/index.html#banner, 2015.

[32] J. Edvardsson, "A Survey on Automatic Test Data Generation," in *Proceedings of the Second Conference on Computer Science and Engineering in Linkoping*, 1999, pp. 21–28.

[33] "Wireshark," www.wireshark.org.

[34] S. Labs, "Appium," http://appium.io/, 2016.

[35] Android, "UI Automator Viewer," http://developer.android.com/tools/testing-support-library/index.html#UIAutomator, 2016.

[36] S. Hao, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps," in *The 12th International Conference on Mobile Systems, Applications, and Services, Mobisys*. ACM, 2014.

[37] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Toch-Sensitive Record and Replay for Android," in *The 35th International Conference on Software Engineering, ICSE*. IEEE, 2013, pp. 72–81.

[38] A. Inc, "Testflight," https://www.testflightapp.com/, 2016.

[39] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin, "AMC: Verifying User Interface Properties for Vechicular Applications," in *The 11th International Conference on Mobile Systems, Applications, and Services, Mobisys*. ACM, 2013, pp. 1–12.

[40] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *2th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM*. ACM, 2012, pp. 93–104.

[41] L. L. Zhang, C.-J. M. Liang, W. Zhang, and E. Chen, "Towards a contextual and scalable automated-testing service for mobile apps," in *The 18th International Workshops on Mobile Computing Systems and Applications (HotMobile)*. ACM, 2017, pp. 97–102.

[42] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *The 9th International Conference on Mobile Systems, Applications, and Services, Mobisys*. ACM, 2011, pp. 239–252.

[43] L. Li, A. Bartel, T. F.Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *The 37th International Conference on Software Engineering, ICSE*. ACM, 2015.

[44] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: Automated Security Validation of Mobile Apps at App Markets," in *Proceedings of the second international workshop on Mobile cloud computing and services, MCS*, 2011, pp. 21–26.

[45] D. Qcteau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *USENIX Security Symposium, Security*. USENIX, 2013, pp. 543–558.

[46] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps," in *The 10th International Conference on Mobile Systems, Applications, and Services, Mobisys*. ACM, 2012, pp. 267–280.