2017 IEEE/ACM 39th International Conference on Software Engineering

Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments

Siegfried Rasthofer, Steven Arzt
Fraunhofer SIT and TU Darmstadt
Germany

Stefan Triller
Fraunhofer SIT
Germany

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

***Abstract*—Android applications, or apps, provide useful features to end-users, but many apps also contain malicious behavior. Modern malware makes understanding such behavior challenging by behaving maliciously only under particular conditions. For example, a malware app may check whether it runs on a real device and not an emulator, in a particular country, and alongside a specific target app, such as a vulnerable banking app. To observe the malicious behavior, a security analyst must find out and emulate all these app-specific constraints. This paper presents FuzzDroid, a framework for automatically generating an Android execution environment where an app exposes its malicious behavior. The key idea is to combine an extensible set of static and dynamic analyses through a search-based algorithm that steers the app toward a configurable target location. On recent malware, the approach reaches the target location in 75% of the apps. In total, we reach 240 code locations within an average time of only one minute. To reach these code locations, FuzzDroid generates 106 different environments, too many for a human analyst to create manually.**

I. Introduction

Android is the most popular platform for mobile devices[1] and there are millions of applications, or apps. Unfortunately, the platform attracts not only benign but also malicious apps. Such malware tries to steal the user's sensitive information or money, e.g., by sending SMS messages to costly premium-rate numbers without the user's consent.

To avoid being detected as malware through automated or manual analysis, many malware apps exhibit their maliciousness only when being executed in a particular environment. For example, some apps check whether they are running in an emulator or another analysis environment, and behave benignly in these cases. Other malware apps target specific countries and remain harmless unless the SIM card in the victim's phone is registered in one of the target countries. Yet another kind of malware targets devices with a specific app installed, such as a vulnerable banking app.

These environment dependencies make it hard for dynamic analyses, such as TaintDroid [13], DroidScope [42], and AppsPlayground [34] to detect malware because an arbitrary execution is likely to be benign. While static analyses, such as AndroidLeaks [16], FlowDroid [4] or DroidSafe [19] do not have this limitation, they usually suffer from false positives and must be complemented by dynamic approaches or manual investigation. A human security analyst then faces the same challenges as a dynamic analysis, e.g., when debugging an obfuscated app with non-trivial environment dependencies. Due the large number of environment properties in Android (SIM country code, mobile network code, location, presence of apps and files on the phone, etc.), it is practically infeasible to simply try all possible environments.

Recent work [40] proposes to generate input for malware apps by modifying the values provided by the device to the app. Unfortunately, the approach relies on a limited set of techniques for finding such inputs and does not support apps containing anti-static analysis techniques. As a result, the approach fails to generate an appropriate environment for many current malware apps. Furthermore, the approach involves significant human effort, making it unsuitable for automatically analyzing a large number of apps.

This paper presents FuzzDroid[2], a framework for automatically generating an Android execution environment where an app exposes its malicious behavior. The main idea is to fuzz the values that the app obtains when interacting with its environment through an extensible set of static and dynamic analyses that provide possible values. FuzzDroid chooses among these values using an evolutionary algorithm-based search that leads the app to a specified target location. If successful, the approach reports an environment sufficient to reach the target location. For example, a security analyst may specify as target all locations that trigger potentially malicious behavior, such as sending SMS messages or aborting incoming SMS messages. The environment reported by FuzzDroid enables the analyst to understand the maliciousness of the app. Furthermore, it enables a dynamic analysis to further investigate the app's behavior, e.g., for automated classification of apps.

We evaluate FuzzDroid with 209 state-of-the-art malware apps. The approach effectively reaches a target location in 75% of these apps, while requiring only one minute per target location, on average. To reach these targets, FuzzDroid generates 106 different environments, which contain four different values, on average. These results show that simply guessing arbitrary combinations of environment values is very unlikely to reach a specific target.

In summary, this paper contributes the following:

- A multi-analysis framework to generate Android environments that expose otherwise hidden malicious behavior.

[1] http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37

[2] Source code available at https://github.com/srasthofer/FuzzDroid

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.35

```
1  String hashValue = "389d90db090f0f0303030030d98676ie03"
2  // called when an SMS message arrives
3  public void onReceive(Context c, Intent intent) {
4    String smsMessage = intent.getMessageBody();
5    // checks content of SMS message
6    if (smsMessage.startsWith("ak40_1") {
7      String certificateHash = this.getCertificate().toString();
8      // integrity check whether the APK got modified
9      if (certificateHash.equals(hashValue)) {
10       // get MCC and MNC codes
11       String mobileOperator = getNetworkOperator();
12       File encryptedFile = readFileFromStorage();
13       File decryptedFile = decryptFile(encryptedFile);
14       boolean containsMobileOp = false;
15       Reader bf = new Reader(decryptedFile);
16       String line;
17       // checks whether file contains mobile operator
18       while((line = br.readLine()) != null) {
19         if(line.equals(mobileOperator)) {
20           containsMobileOp = true;
21           break;
22         }
23       }
24       // targeted attack against specific network
25       if (containsMobileOp) {
26         // dynamic class loading, expects a dex file,
27         // even though file suffix is .db
28         DexClassLoader dcl = new DexClassLoader("anserverb.db");
29         Class clazz = dcl.loadClass("BaseABroadcastReceiver");
30         Method method = clazz.getMethod("onStart", Intent.class);
31         boolean returnValue = (boolean) method.invoke(intent);
32         if (returnValue == false) {
33           // target location: aborts delivery message
34           this.abortBroadcast();
35 }}}}}
```

Fig. 1. Motivating example: Incoming SMS messages are aborted only under certain circumstances.

- A set of static and dynamic analyses that provide values for circumventing various checks in malware apps.
- A search-based fuzzing algorithm that selects environment values to steer an app toward a target location.
- Empirical evidence that the approach is efficient and effective for current malware apps, and that it clearly outperforms the closest existing approach.

II. Overview and Example

This section illustrates several key challenges for reaching a particular target location in a state-of-the-art malware app and outlines how we address these challenges. Figure 1 shows a motivating example, which we reverse-engineered from several real-world Android malware apps from the *Anserver*, *FakeInstaller* and *BadAccents* malware families. The app intercepts SMS messages sent to the device and, under particular conditions, aborts the delivery of the message to the user's inbox. Malware apps use such behavior, e.g, to intercept validation messages sent to bank customers. The malware then silently uses the validation code to perform malicious transactions.

To understand an app's behavior, a human analyst or an automated analysis is interested in the conditions under which the app aborts the delivery of messages. In the example, suppose that we mark the call to the `abortBroadcast` method (line 34) as the target location and want to trigger an execution where the app calls this method. Reaching this target location is difficult because the app requires a particular environment to expose its malicious behavior:

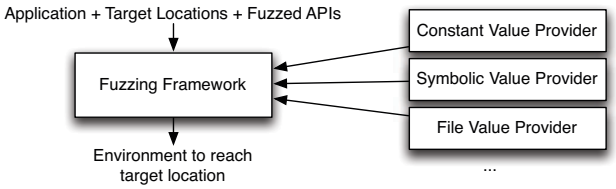

Fig. 2. Overview of the FuzzDroid approach.

- The behavior must be triggered by an incoming SMS message that starts with a particular string (line 6).
- The app checks whether the user's network operator is part of a pre-defined list of targets (lines 11 to 25).

The problem is compounded by the fact that the app tries to evade analysis:

- The app checks whether the APK file of the app has been modified by comparing its signature against a known value (line 9). Such checks aim at preventing instrumentation-based dynamic analyses from modifying the app.
- The app loads an additional class from a file, reflectively calls a method of the loaded class, and checks whether the method returns a particular value (lines 28 to 32). Such behavior challenges static analyses, which cannot easily reason about reflective methods calls, and in particular, about methods of classes that are dynamically loaded from an external file.

All these conditions are typically hidden in highly obfuscated code. A naive dynamic analysis that simply executes the app, possibly by sending a random SMS, does not reach the target location. Randomly fuzzing the environment is very unlikely to be successful, because multiple non-trivial conditions must be met. A purely static analysis cannot easily reason about reflectively called methods, and in particular, about methods of classes that are dynamically loaded from an external file. Even a recently-proposed approach that combines static and dynamic analysis [40] cannot reach the target, partly because it cannot deal with obfuscation through reflection.

Figure 2 gives a high-level overview of the FuzzDroid approach. Given an app, a set of target locations, and a set of APIs to fuzz, the approach repeatedly executes the app while fuzzing the values returned by the specified APIs, until it finds an environment where the app reaches a target location. To this end, the approach intercepts calls of the app to APIs and modifies their return values to steer the app toward the target.

An observation crucial for our work is that a single fuzzing approach is insufficient to reach target locations in real-world malware apps. Instead, FuzzDroid consists of a generic framework and an extensible set of value providers. Each *value provider* is a static or dynamic program analysis that provides values to the fuzzed APIs. To decide which of the suggested values to use for a particular API call, we present an evolutionary algorithm-based search strategy. The strategy

iteratively refines the selection of values based on feedback from previous executions.

To illustrate FuzzDroid with the motivating example, suppose we trigger the `onReceive` method with an empty SMS message. At line 4, the approach intercepts the first call to the environment, `getMessageBody`, and queries multiple analyses for possible return values. Suppose that one analysis, which extracts string constants from the app's bytecode, suggests the value "anserverb.db". Another analysis, which reasons about string operations by extracting and solving constraints, suggests a value "ak40_1abc". Suppose that FuzzDroid randomly decides to return the value "anserverb.db", so the `onReceive` method returns without reaching the target.

Next, the approach re-executes the method and again reaches the fuzzing decision at line 4. Suppose that now, FuzzDroid fuzzes the `getMessageBody` call by returning "ak40 _1abc", so the app takes the if branch and gets closer to the target location. During all executions, the approach keeps track of the distance of the executed path to the target location and exploits this knowledge to prioritize values. For the example, the approach may infer from the first two executions that "ak40_1abc" leads the execution closer to the target, and is thus more likely to reach the target than "anserverb.db".

To eventually reach the target, FuzzDroid proceeds to fuzz environment calls while improving the selection of fuzzed values, until it finds suitable return values for further environment calls at lines 7, 11, 28, and 31. Key to reaching the target is to combine values extracted with several, complementary analyses, instead of relying on a single analysis. Sections III and IV present our approach in more detail.

III. A Targeted Fuzzing Framework

The FuzzDroid approach consists of a generic fuzzing framework and an extensible set of value providers for fuzzing particular APIs. This section presents the fuzzing framework. At first, we present the overall algorithm of the framework (Section III-A). Then, we describe how the framework interacts with the app during the execution (Section III-B). Next, Section III-C presents how the framework steers the execution toward a target location by picking appropriate values for the fuzzed APIs. Finally, we present how FuzzDroid deals with dynamically loaded code (Section III-D).

The goal of FuzzDroid is to find an environment in which the app reaches a target location. An Android app interacts with its environment through API calls, such as `getDeviceId` and `getMessageBody`. We control the environment of an app by fixing the values returned by such API calls:

Definition 1 (Environment) *An environment $\mathcal{E} : \mathcal{L} \times \mathbb{N} \to \mathcal{V}$ is a map that assigns a value $v \in \mathcal{V}$ to a pair (l, n), where $l \in \mathcal{L}$ is a code location and $n \in \mathbb{N}$ is a counter of how often l has been reached in the current execution.*

For example, suppose that l_{SMS} is a call site of `getMessageBody`. The environment $\{(l_{SMS}, 1) \mapsto "abc", (l_{SMS}, 2) \mapsto "def"\}$ specifies that the call to `getMessageBody` returns "abc" and "def" when the location is reached for the first and second time, respectively. We call an environment that enables the app to reach a target location a *successful environment*. Note that we only consider a target location as reached if it can be executed without an exception.

Algorithm 1 Find an environment $\mathcal{E}$ that reaches location l_{target} in app.

Input: App app, set $\mathcal{L}_{target}$ of target locations, and set $\mathcal{A}_{fuzz}$ of APIs to fuzz
Output: Environment $\mathcal{E}$

```
1:  Q ← [app]                                              ▷ app queue
2:  while Q ≠ empty do
3:      app_current ← Q.pop()
4:      staticPreAnalysis(app_current)
5:      instrument(app_current, L_target, A_fuzz)
6:      nbRuns ← 0
7:      T ← ∅                                              ▷ trace pool
8:      while nbRuns < maxRuns do
9:          nbRuns ← nbRuns + 1
10:         E ← initializeEnvironment(T)
11:         trace ← executeAndFuzz(app_current, E, Q)
12:         T ← T ∪ {trace}
13:         if targetReached(trace, L_target) then
14:             report(E, trace)
15:             exit
16:         end if
17:     end while
18: end while
```

A. Main Algorithm

To find a successful environment, FuzzDroid repeatedly executes the app while refining the environment, as summarized in Algorithm 1. The outer loop of the algorithm will be explained in Section III-D; the reader should ignore it for now and focus on the steps starting from line 3.

At first, the algorithm statically builds an inter-procedural control flow graph of the app, which will be used by the subsequent steps. In addition, each value provider plugged into the framework can perform further static analyses at this point. Next, the framework instruments the app to keep track of the execution path and to intercept calls to the fuzzed APIs.

The main loop of the algorithm starts at line 8. The framework repeatedly executes the app until either a target location or a configurable maximum number of executions has been reached. Before each execution, function $initializeEnvironment$ creates an environment. Section III-C describes this step in detail. During the execution, the instrumented app queries the framework for values to be returned at call sites of fuzzed APIs. Section III-B presents the $executeAndFuzz$ function, which implements this step, in detail. The framework summarizes each execution into an execution trace and maintains a pool of all previous executions. After each execution, the algorithm checks whether the

target location has been reached (line 13). If the framework has reached the target, the algorithm returns the successful environment. Otherwise, the algorithm refines the environment based on the feedback obtained from previous executions and executes the app again.

B. Executing and Fuzzing Apps

The core of FuzzDroid's fuzzing happens in function $executeAndFuzz$, called at line 11 of Algorithm 1.

1) Fuzzing the Environment: Each execution starts with an initial environment $\mathcal{E}$ that maps a subset of all possible API calls that may happen during the execution to return values. During the execution, the app queries the framework whenever the execution reaches a fuzzed API. If the app requires a pair $(l, n) \in \mathcal{E}$, i.e., a value provided by the initial environment, then the framework returns this value. Otherwise, the framework queries the value providers, selects one of the provided values, and adds this value to the environment $\mathcal{E}$.

To help the framework select a value, value providers associate with each value a weight that specifies the confidence the value provider has in the respective value. The weights also allow for prioritizing particular value providers over others, e.g., if one value provider is generally more precise than others. The framework selects a value by ordering all provided values and by picking randomly among the values with the highest weight. To prevent the framework from permanently rejecting values with low weight, the framework also considers all remaining values with a low probability (10% per default) and selects randomly among them, regardless of their weight.

During the execution of the app in the fuzzed environment, the framework summarizes the execution into a trace:

Definition 2 (Trace) *A trace $t = (L, \mathcal{E})$ summarizes the execution of an app into the list $L = [l_1, ..l_n]$ of executed code locations $l_i \in \mathcal{L}$ and the environment $\mathcal{E}$ that has triggered this execution.*

At the end of an execution, the framework adds the trace to a *trace pool*. These traces have two purposes. First, the framework creates future initial environments based on the traces of previous executions, as described in detail in Section III-C. Second, value providers can adjust the set of provided values based on the current trace and on the trace pool. For example, a value provider may reduce the weight based on values already used in previous executions or provide values based on the path taken in the current execution.

2) Triggering Events and Services: The approach described so far assumes that the target location is reachable by simply starting the app under a suitable environment. However, some target locations may only be reached when the app reacts to a particular event, such as an incoming SMS message or a click on a button. To enable the approach to reach such target locations, FuzzDroid programmatically triggers event handlers. For this purpose, the framework computes a static call graph of the app and traverses it backwards, starting at the target location. When reaching the beginning of a callback method, FuzzDroid checks the event for which the respective callback is registered and triggers the event programmatically. We directly call the respective event handler method and thus do not need a model of the entire app and its UI. Even if several UI interactions would be necessary to trigger the event during normal execution, FuzzDroid directly jumps into the handler.

3) Properties of Successful Environments: Since FuzzDroid checks dynamically whether an environment reaches the target location, a reported environment is guaranteed to reach the target. In contrast, FuzzDroid guarantees neither to find a minimal environment nor to generate a realistic environment, i.e., a reported environment may over-constrain individual environment values, possibly with values that may not occur in reality. Suppose an app requires the name of the network operator to contain the string "tele". In this case, FuzzDroid may report an environment that sets the network operator to "teleFoo", which is not an actual network operator. Our evaluation shows that the absence of these guarantees is a non-issue in practice, because most successful environments specify a manageable number of values.

C. Steering Toward the Target

Since the set of possible environments that FuzzDroid can generate is too large to explore exhaustively, it is crucial to steer the approach toward an environment that reaches the target location. This section explains how FuzzDroid steers toward such an environment based on the trace pool. These steps of the approach correspond to function $initializeEnvironment$ in Algorithm 1.

1) Measuring the Fitness of Environments: To identify environments that are likely to lead the app to the target location, we compute a fitness score for each environment based on the trace that the environment yields:

Definition 3 (Fitness of an environment) *Given a trace $(L, \mathcal{E})$ and a target location l_{target}, the fitness of $\mathcal{E}$ is the minimum distance between l_{target} and any location l in L.*

FuzzDroid computes the distance between code locations as the minimum number of edges between the locations in the inter-procedural control flow graph. The rationale for considering the minimum distance is that traces that get close to the target at some point during the execution are more likely to reach the target than traces that always remain far from the target. Other measures can be easily added to our approach.

2) Evolutionary Algorithm: Based on the fitness of environments, an evolutionary algorithm creates new initial environments. The intuition behind the approach is that traces which came close to the target are likely to have values in their environment that enable the app to reach the target. The algorithm repeatedly combines such promising environments into new environments, until FuzzDroid reaches the target.

Algorithm 2 summarizes the main steps. Given a trace pool $\mathcal{T}$, the algorithm computes an environment $\mathcal{E}$ for the next execution. At first, the algorithm checks whether the

Algorithm 2 Create an initial environment $\mathcal{E}$.

Input: Trace pool $\mathcal{T}$
Output: Environment $\mathcal{E}$

```
1: if |T| < minTraces or randomNb() < 0.25 then
2:    return empty environment
3: end if
4: E_sorted ← sortEnvsByFitness(T)
5: while E_sorted ≠ empty do
6:    E_1, E_2 ← selectParentEnvs(E_sorted)
7:    E ← crossover(E_1, E_2)
8:    if isNewEnv(E, T) then
9:       mutate(E)
10:      return E
11:   else
12:      remove(E_sorted, E_1)
13:   end if
14: end while
15: return empty environment
```

size of the trace pool exceeds a minimum value (five in our evaluation), and otherwise, returns an empty environment. The empty environment forces FuzzDroid to query value providers at runtime for values, which yields additional traces to learn from. If there are sufficiently many traces, the algorithm sorts the environments of these traces by their fitness score. Next, the main loop of the algorithm (lines 5 to 14) performs the following classical steps of an evolutionary algorithm: select two parent environments, create a new environment through a crossover operation, and check whether this yields an environment that has not yet been used in any previous execution. If so, the algorithm mutates the new environment and returns it. Otherwise, the search continues until the algorithm runs out of possible parent environments. In this case, the algorithm returns an empty environment.

a) Selecting Parents: The $selectParentEnvs$ function (line 6) selects the two environments with the highest fitness. If combining these environments yields an environment that has already been tried before, then the algorithm removes the current best environment from the sorted list E_{sorted} of environments to consider. As a result, the next iteration of the algorithm's main loop selects the second-best and third-best environment as potential parents, and so on, until there are no more possible parent environments. Note that the sorted list E_{sorted} from line 4 is a copy of the original trace pool $\mathcal{T}$. Therefore, the removal of the environment does not influence the overall trace pool $\mathcal{T}$.

b) Crossover: To combine two environments, FuzzDroid first computes the union of all keys in the environments. If a key is provided by only one environment, then this environment contributes the values. Otherwise, FuzzDroid randomly picks which environment contributes the value.

c) Mutation: To avoid getting stuck in a local minimum, i.e., an environment that brings the execution close to the target but based on which the execution cannot reach the target, the algorithm mutates the environment that results from crossover. FuzzDroid mutates each value of an environment with a small probability (0.1 in our evaluation). To mutate a value for a particular API method, the approach picks from an environment different from the two current parents a random value provided for this API method.

D. Dealing with Dynamic Code Loading

Some malware apps hide malicious behavior by storing the malicious code in an encrypted file and by decrypting and loading this code at runtime. In this case, the malicious code is unavailable to the static part of our framework. In particular, the target location may not be visible to static analysis, making it impossible to, e.g., compute the distance between the target and already executed locations. FuzzDroid deals with such *packing* by observing dynamically loaded code and by rewriting the app into an app that contains this code. To this end, FuzzDroid first takes all locations of dynamic code loading as targets and attempts to steer the execution to these parts of the code. When reaching a location that dynamically loads code, the framework merges this code with the original APK file, enabling future runs to analyze the full, not obfuscated code. In Algorithm 1, the list Q represents the queue of apps created by merging apps with dynamically loaded code.

To call methods from dynamically-loaded code, apps use the Java reflection API. As a further obfuscation step, the method and class names often are computed or decrypted at runtime, making them unavailable to static analysis even if all code is available. To address this challenge, FuzzDroid applies Harvester [32] to each app. Harvester uses a combination of static and dynamic analysis to precisely extract the targets of reflective method calls and replaces them with direct calls.

IV. Value Providers

This section presents value providers, which create values to be returned at call sites of fuzzed APIs. A main contribution of FuzzDroid is to automatically combine multiple value providers.

A. Symbolic Value Provider

The way an app uses the values obtained from the environment often reveals the app's expectations about these values. For example, line 6 in Listing 1 reveals that the incoming message must start with `ak40_1`. Likewise, a call to `sendTextMessage(nr, body)` reveals that `nr` is expected to be a number or to start with a "+", e.g., "+491234". To exploit such information, FuzzDroid contains a constraint-based, symbolic analysis that reasons about uses of environment values in the app. The basic idea is to encode the results of a local, static analysis and values extracted at runtime into constraints, and to query a constraint solver to find values for fuzzing. The analysis computes for each call site $\mathcal{L}_{fuzz}$ of a fuzzed API a set of values. The approach consists of three steps explained in the following.

1) Static data flow analysis: The approach statically reasons about the uses of fuzzed values through an interprocedural data flow analysis. Given a set of source locations and a set of sink locations, the analysis extracts sequences of statements that propagate and modify values between a source and a sink. We call such a sequence of statements a *data flow path*. The sources for our analysis are all call sites of fuzzed APIs. As sinks, we consider all call sites of a configurable set of methods that reveal expectations by the app on values, such as `String.startsWith` and `sendTextMessage`. The static analysis yields a map $\mathcal{L}_{fuzz} \rightarrow \mathcal{D}$ from call sites of fuzzed APIs to sets of data flow paths.

For example, Listing 1 yields a data flow path that connects `getMessageBody` via `smsMessage` with `startsWith("ak40_1")`.

2) Constraint solving: The analysis translates the data flow paths into constraints and solves them to obtain values to be suggested at call sites of fuzzed APIs. At first, the approach transforms each data flow path into SSA-form. Next, the approach translates data flow paths into conjuncts of constraints understood by the Z3 solver [12], similar to prior work [5]. Specifically:

- The initial call of a fuzzed API method in each data flow path is represented as a symbolic variable v_{fuzz}.
- Any operations applied in the data flow path are translated into the corresponding Z3 constraints. For example, we translate string operations into their corresponding constraints provided by the Z3 string theory [44]. For API methods not supported by the Z3 solver such as `String.split`, we provide additional constraints that model the behavior of the respective API.
- To encode the information revealed by the call of the sink method, we translate this call into constraints. For example, a call to `sendTextMessage` is encoded as constraints that specify that the first argument provided to the method must be a number or "+" followed by a number.
- For data flow paths with sink methods that represent boolean checks, such as `String.equals`, we generate two conjuncts of constraints, which represent the case that the check returns true and false, respectively.

This translation yields a map $\mathcal{L}_{fuzz} \rightarrow \mathcal{C}$ from call sites of fuzzed APIs to sets of symbolic constraints. To compute values for fuzzing, the approach queries the solver for each set of constraints to obtain a concrete value for v_{fuzz}. The solving yields a map $\mathcal{L}_{fuzz} \rightarrow \mathcal{V}$ that assigns to each fuzzed location a set of possible values. Whenever the framework queries the analysis for a value to be returned at a location $l \in \mathcal{L}_{fuzz}$, the analysis returns one of the possible values. To reduce the computational cost of constraint solving, the analysis solves all statically extracted constraints before executing the app for the first time.

3) Dynamic refinement of constraints: The statically extracted constraints may contain symbolic variables in addition to the fuzzed value v_{fuzz}. For example, suppose an app compares the return value of a fuzzed API to a dynamically created string using `String.equals`. Without knowing the dynamically created string, the constraint solver is unlikely to return a suitable value for v_{fuzz}, because it knows only that v_{fuzz} is equal to another symbolic variable v_c. We address this problem by enriching the statically computed constraints with dynamically extracted values. To this end, the analysis obtains from the framework runtime values involved in calls of a configurable set of methods. By default, we include into this set of methods string operations, such as `String.equals` and `String.substring`, because these operations are particularly important in various malware apps. If an execution produces a concrete value c for a symbolic variable v_c in one of the statically computed constraints, then the approach copies the constraint and, in the copy, replaces the v_c with c. By solving the refined constraints, the analysis is more likely to obtain a suitable value for v_{fuzz}. For the above example, the refined constraints specify that v_{fuzz} is equal to c, making it trivial to find a precise solution.

To reduce the cost of constraint solving, the analysis performs the dynamic refinement of constraints on demand. That is, whenever the framework queries the analysis for a value at a location, the analysis checks whether the constraints for this location contain any symbolic variable for which concrete runtime values have been observed in previous executions. Only if such runtime values exist, the analysis gives the refined constraints to the solver. Since the results computed from constraints that make use of dynamic values are usually more precise than those that rely only on static data flow paths, the approach gives them a higher weight when providing them to the framework.

Our constraint-based analysis differs from traditional symbolic and concolic execution [8], [9], [17], [18], [22], [36], [37] by applying a local symbolic analysis instead of reasoning about the entire execution path. The benefit of avoiding path sensitivity is that our local analysis scales well to large apps. However, the approach cannot guarantee that values obtained from the solver will cause the app to reach the target. For example, our analysis does not reason about which of the two branches of a conditional leads to the target, but instead, suggests values for both branches. Since FuzzDroid dynamically validates whether an environment reaches the target, the symbolic analysis does not have to provide this guarantee. Instead, FuzzDroid iteratively selects values based on the fitness of executed environments (Section III-C).

B. Constant Value Provider

Many apps compare runtime values against constants stored in the code. These constants may not be directly in the conditional but, e.g., read from variables or fields. To execute branches guarded by such conditionals, a simple static analysis gathers from an app's bytecode all constants of primitive types and strings. The value provider returns these constants when being queried for a value of a matching type. If the app does not contain any statically extractable constants, e.g. due to obfuscation, the analysis returns values from a pre-defined pool of random values.

For the example in Figure 1, the constant value provider helps pass by the integrity check at line 9. The value provider extracts the hash value of the certificate at line 1 and suggests it when the app queries the certificate hash at line 7. As part of our future work, we plan to also take dynamically-computed values into account, a feature that is already implemented in the symbolic value provider (see Section IV-A).

C. File Value Provider

For some malware apps, the existence of a file, possibly containing data of a particular file type, is essential for triggering malicious behavior [33]. To prevent apps from failing when an expected file is missing, the file value provider suggests values for APIs that access the file system. If the accessed file does not exist, the provider emulates the file. To this end, the provider infers the expected file type and provides a dummy file of the inferred type.

During the static pre-analysis, the analysis approximates the set of expected file types using a forward data flow analysis that follows the data flow from the file access to an API call that reveals the expected file type. For example, a dataflow that reaches a `SQLiteDatabase.openOrCreateDatabase` calls reveals that the file is expected to be a database file. We provide a manually assembled map between API calls and file types. Once the type is known, a manually created dummy file of the correct type is picked and pushed onto the phone before the app accesses that file. If the analysis fails to statically identify the expected file format, it tries to create a suitable file based on the name of the accessed file.

Note that this analysis aims at providing a file with the expected format, not a file with the expected content. Further API calls that, e.g., read data from the file, are intercepted separately. Other analyses such as the symbolic value provider (Section IV-A) can then provide the values expected to be read from the file.

D. Value Provider for Integrity Checks

Many malicious apps protect against code modifications by validating their own integrity through a check of the app's certificate, which is used for signing an Android app. When modifying the app's bytecode, one must re-sign the app. However, without access to the original developer's private key, it is practically impossible to use the same certificate for signing the app. A common way to implement an integrity check is to compute the hash code of the signature certificate extracted from Android's package manager and to compare it to an expected value. Alternatively, an app may also use the computed hash to decrypt data, e.g., dynamically loaded code.

To circumvent such checks, FuzzDroid contains a value provider that fuzzes API calls that access an app's signature. Instead of the real certificate of the actually running app, the value provider returns the certificate that was used to sign the original, uninstrumented app, effectively fooling the check.

E. Primitives-as-Strings Value Provider

Java supports various API calls for data type conversion, e.g., to convert a string into a numeric value via `Integer.valueOf`. Such conversions give hints on the expected format of unknown data. The most common case are boolean flags that malware apps reads from Android's shared preferences as strings "true" or "false". These strings are then converted to boolean flags that may determine whether the target location executes. Most of the time, the behavior is disabled by default. Only upon, e.g., a request from a remote command-and-control server, the app enables the flag in the file, i.e., changes the file contents to "true" in order to enable the malicious behavior.

The primitives-as-strings value provider uses static data flow analysis to track data flows from code locations where values are obtained (e.g., file reads or the `Properties.getProperty` method) to data type conversion methods. If the analysis is able to identify the type of the primitive, the value provider picks a value from a pre-defined set of random values of the correct type.

V. Implementation

We use Soot [39] with the Dexpler front-end for Android [7] to statically analyze and dynamically instrument the apps. For implementing data flow analyses, we build upon FlowDroid [4]. The FuzzDroid framework runs on a desktop computer while the app runs on an Android emulator. Both communicate via a TCP connection, e.g., to request environment values or to report which path is executed. To intercept fuzzed API calls, we use a user-space hooking library (ZHookLib[3], based on the Xposed framework[4]).

Some malware apps contain timing bombs, i.e., actions that are only executed after a certain time. To avoid having to wait for this time span, we statically patch those statements during our instrumentation phase and decrease the waiting time to a few seconds.

VI. Evaluation

We evaluate the effectiveness and efficiency of FuzzDroid by applying it to 209 malicious Android apps. Our evaluation focuses on the following research questions:

- How effective is the approach at finding an environment that enables an app to reach a target location?
- How effective are FuzzDroid's use of multiple analyses and the way these analyses are combined with each other?
- How efficient is the approach?
- What do the environments generated by the approach reveal about real-world malware?
- How does FuzzDroid compare to the best existing approach for generating inputs that steer an Android app toward a particular location?

A. Experimental Setup

We randomly collected 300 recent malware apps from Virustotal[5] in June 2016. As target locations, we use call sites of seven API methods related to SMS messages, e.g.,

[3]https://github.com/cmzy/ZHookLib
[4]http://repo.xposed.info/
[5]Online malware database: https://www.virustotal.com/

	Approach		
	Launch	Launch & trigger	Fuzz-Droid
Apps with ≥ 1 target reached	31.28%	45.02%	75.12%
Target locations reached	10.48%	15.82%	62.34%
When target reached (min/avg/max):			
- Executions	1/1/1	1/1.33/6	1/3.20/14
- Time to target (seconds)	3/8/18	3/19/310	3/62/1469
- Size of environment	—	—	0/3.69/47
- Contributing analyses	—	—	0/1.14/4

TABLE I
OVERVIEW OF RESULTS. FOR VALUES SUMMARIZED OVER MULTIPLE APPLICATIONS, WE PROVIDE THE MINIMUM/AVERAGE/MAXIMUM VALUES.

`sendTextMessage`. Sending SMS messages and aborting incoming SMS messages are a common threat for Android applications [3], [29]. Out of the 300 apps, 209 apps contain at least one target location. As environment APIs, we use 32 API methods. We run FuzzDroid on a server with 64 Intel Xeon E5-4650 CPUs running at 2.70 GHz and 1 TB of physical memory. We configured FuzzDroid to run at most 15 executions and to start the genetic recombination after 5 runs.

B. Effectiveness in Reaching a Target Location

We evaluate the effectiveness of FuzzDroid at finding an environment where the application reaches a target location. Furthermore, we compare the approach to two simpler approaches: 1) simply launch the app and hope that it will reach the target location without further intervention, and 2) launch the app and trigger specific events, such as clicking a button or sending an SMS message, as described in Section III-B2. In neither of the two simpler approaches, we generate a particular environment. Instead, if the app calls environment APIs, the emulator's default values are returned.

Table I shows our results. We find that running the app under the "right" environment is crucial for reaching the target location. The default environment of the emulator is insufficient for most current malware. Furthermore, the results show that FuzzDroid is effective in generating an environment that successfully reaches the target location. In total, the approach reaches 240 different target locations (62.34%).

In some cases, FuzzDroid fails to reach the target location. For example, several malware apps contain malicious behavior which is, however, not yet enabled, and thus never called. We conjecture that the app will be updated at some point to actually activate the malicious behavior that is already present in the code.

C. Importance of Multi-Analysis Approach

To evaluate how much the individual analyses of FuzzDroid contribute to the framework's overall effectiveness, we first evaluate the framework with all analyses enabled. Then, we disable each analysis in turn, i.e., run the framework with all but one analysis and then run it with a single analysis. The result indicate how much the effectiveness decreases if this analysis is left out. The results in Figure 3 show that there is no single analysis that can be left out without a negative impact, i.e., all analyses are necessary. This shows that only a framework supporting multiple interacting analyses such as FuzzDroid is able to find correct execution environments for state-of-the-art malware.

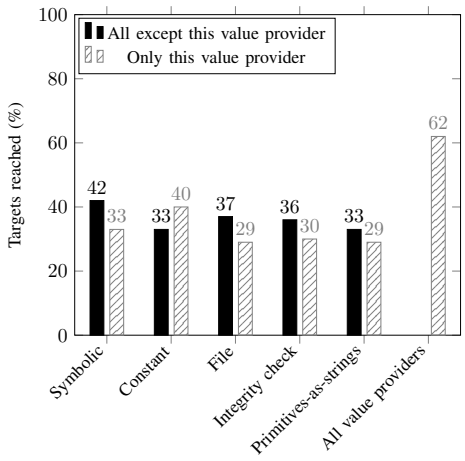

Fig. 3. Effectiveness with different subsets of all value providers.

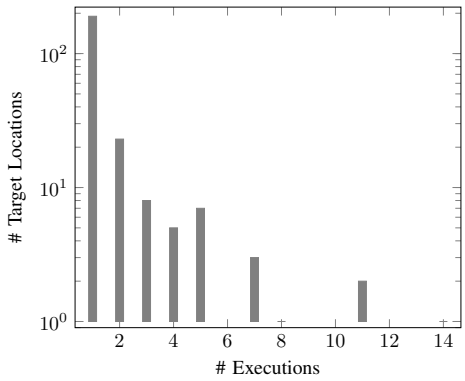

Fig. 4. Executions required to reach target location.

D. Efficiency

Table I shows how long FuzzDroid takes to find an environment under which the application reaches the target location. On average, it takes 62 seconds to reach a target location. Most of the time (75%, 46.5 seconds) is spent for executing the app. In contrast, both instrumenting apps (5%, 3.1 seconds) and statically analyzing apps (20%, 12.4 seconds) are secondary for the overall time.

Figure 4 shows the number of executions required by FuzzDroid to reach the target location. In many cases, the target location is reached in a single execution, i.e., the first generated environment is sufficient. For a substantial number of apps, however, the approach tries two to seven environments, showing that incremental fuzzing is required to reach the target location. The maximum number of executions needed during our evaluation is 14.

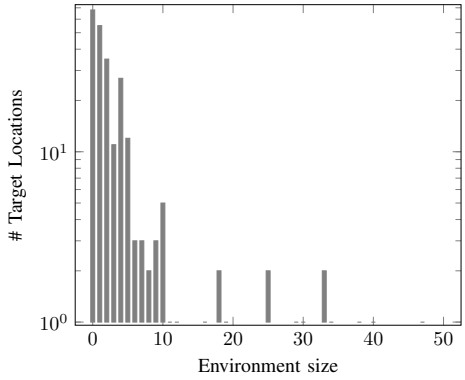

Fig. 5. Size of environment to reach target location.

Kind of environment values	Prevalence
File access	47.97%
SIM/network operator code	16.82%
Specific incoming SMS message	10.84%
SIM operator name	5.53%
Timing bomb	4.06%
SIM country	3.16%
Integrity check	1.02%
Admin check	0.68%
Others	9.92%

TABLE II
PREVALENCE OF DIFFERENT KINDS OF ENVIRONMENT VALUES.

E. Environments Generated by FuzzDroid

1) Number and Size of Environments: The environments required to reach a target location range from trivial environments, where simply starting the app in the default emulator is sufficient, to complex environments with dozens of values. Across all apps where a trivial environment is insufficient, FuzzDroid creates 106 different successful environments, which would be impractical to create manually by a human analyst.

Figure 5 shows the sizes of the environments generated by FuzzDroid. For several target locations, no particular environment is required, i.e., the environment size is zero. For most others, two to ten different environment values must be combined to reach the target location, showing that FuzzDroid is highly beneficial for a security analyst interested in triggering malicious behavior. In some cases, the environment even consists of more than 30 values.

2) Examples of Environments: Beyond being useful for security analysts, FuzzDroid allows us to better understand how current malware interacts with the Android environment. The environments generated for several apps show that *targeted attacks* against a particular country, network operator, etc. are common in current malware. Table II summarizes the kinds of values we find in the environments that reach a target location. The following discusses several representative examples.

A very common kind of interaction with the environment is to access information from files. Besides such file accesses, various malware apps target particular SIM/network operators or check the SIM country code. For example, some apps expect SIM operator names to be “mts” or “megafon”, two prominent network providers in Russia, or to match the regular expression “*tele*”, as in “telecom”. We also find several malware apps that target specific countries, either by attacking users in a particular country or by checking that users are not located in a particular country.

Another interesting interaction with the environment are so called *timing-bombs*, where the malicious behavior only gets executed after a specified time has passed. This technique has been crucial for many malware samples to be accepted into the official Google Play Store without being detected [11]. Perhaps surprisingly, relatively few of the apps in our sample check whether the user grants the app device administration privileges.

Packed Malware: During the evaluation we encountered a malware app with an unusually low number of classes. Since all API calls are obfuscated through reflection, the malicious behavior is not directly visible. Instead, the app uses two additional files to hide more code. The original APK file generates a new dex file responsible for unpacking the encrypted malware. The decryption key is derived from the hash of the certificate with which the original APK file was signed. This dependency between certificate and decryption key is supposed to protect the integrity of the malware against, e.g., bytecode instrumentation. After discovering through manual inspection that the malware may show a fake user interface asking about credit card information, we set the `showFakeDialog` call as the target location. FuzzDroid then finds an environment that circumvents the integrity check and makes the malware app believe that the app of “Commerzbank” (a major bank) is opened. Only if both conditions hold, the phishing dialog is shown. This example illustrates (1) that FuzzDroid is able to handle applications with dynamically loaded dex files, (2) what kind of techniques current malware uses for hiding its malicious behavior.

F. Comparison with State of the Art Approach

IntelliDroid [40] is the conceptually closest approach to FuzzDroid. Still, there are important differences. First, IntelliDroid is solely based on constraint solving and does not support multiple analyses. Second, IntelliDroid has a different notion of runtime values. While we consider all values that are dynamically computed inside the app as runtime values, IntelliDroid obtains only the current device state, such as the current time or the registered alarm managers. Only this information becomes part of the constraint system. Therefore, if an app (such as the one in Listing 1) dynamically decrypts data, this data is unavailable to IntelliDroid. Third, IntelliDroid does not intercept API calls made within the app but relies on externally triggered events, such as sending an SMS message. This approach fails for the check on the mobile operator in our example. Fourth, IntelliDroid runs the app under analysis only once and therefore cannot exploit runtime feedback. Finally, IntelliDroid requires a lot of manual interaction with the tool and provides little automation.

Due to the lack of automation, we apply IntelliDroid to a random sample of 20 of our malware apps. We pick apps that

contain at least one target location that is not immediately reached when starting the app or when triggering an event. In total, IntelliDroid reaches 11% of the target locations in the sample, whereas FuzzDroid reaches 62%. We conclude that our approach successfully addresses important limitations of IntelliDroid that prevent the state-of-the-art tool from reaching various target locations.

VII. Limitations

FuzzDroid currently assumes that it is sufficient to trigger a single path to reach the target location. For example, if an app requires a first SMS message to set a flag and then only executes the malicious code when a second SMS message is received after the flag has been set, FuzzDroid cannot trigger the malicious behavior. Analyzing dependencies between multiple events is subject to future work.

Our current research prototype is based on an intra-component callgraph generated with the help of FlowDroid [4]. To detect environment checks that are distributed across multiple components, FuzzDroid must be extended with inter-component analysis tools such as EPICC [30] for callgraph construction or ICCTA [24] for data flows.

VIII. Related Work

a) Malware Analysis: In *X-Force*, Peng et al. [31] propose a binary analysis engine which forces the execution of a program into specific branches. Their approach makes the program agnostic of the execution environment, revealing hidden behavior in malware. GoldenEye [41] exploits several virtual environments executed in parallel. The tool adaptively switches the analysis environment at runtime through a specially designed speculative execution engine. Moser et al. [28] tackles a similar problem for x86 code as we do in our work for Android. They use a dynamic approach in combination with system snapshots of the execution to execute code statements that produce a malicious behavior. To explore different paths, the program state is reset to earlier snapshots. The values on which the conditionals depend are updated to force different branches when the execution is resumed.

Kolbitsch [23] proposes a hybrid approach that combines a light-weight form of static symbolic execution with an instrumentation of additional code statements for a multi-path execution on JavaScript code. Abraham et al. [1] also propose a hybrid approach for reaching a certain target location. Their approach has an success rate of 28% and less on current Android malware and does not report information about the environment. TriggerScope [15] is a pure static approach that relies on symbolic execution for extracting environment information. Being purely static, their approach does not support dynamically loaded code and cannot exploit runtime feedback. FuzzDroid in contrast is a hybrid approach and the generated environments are validated dynamically.

b) Test Generation and Search-based Testing: Symbolic and concolic testing generate inputs by reasoning about path constraints [8]–[10], [17]. In contrast, FuzzDroid applies symbolic reasoning to individual data flow paths, which improves scalability. EvoDroid [25] also uses evolutionary testing for Android apps. Mirzaei et al. [27] create an Android system model in Java Pathfinder [20] to apply symbolic execution to the whole app for increasing test coverage. Thummalapenta et al. [38] generating method call sequences through a combined dynamic-static analysis. Similar to our approach, Malburg and Fraser [26] combine symbolic execution based on Java Pathfinder with a genetic algorithm that negates individual conditions during mutation. All these approaches aim for high coverage, whereas FuzzDroid aims at reaching a particular target location.

Jensen et al. [21] concolically execute events handlers to find sequences of handlers that reach a given target. In contrast to their work, FuzzDroid reasons about the environment of an app, not the order of triggered events. Baars et al. [6] combine symbolic execution with dynamic analysis to improve the efficiency of search-based testing. Applying their fitness function to FuzzDroid is interesting research for future work. Extracting constants from the bytecode is a promising seeding strategy in search-based software testing, as shown by previous research [2], [14]. However, for current Android malware applications, this seeding strategy must be combined with other value providers, as we have shown.

DroidFuzzer [43] also applies fuzzing to Android but creates data for activities that accept MIME data, such as media data. A related approach [35] fuzz tests the messages provided to an app via intents. Both techniques aims at triggering bugs, whereas FuzzDroid provides information for a security analyst.

IX. Conclusions

This paper presents FuzzDroid, a framework for automatically generating an environment under which an Android malware app exposes its otherwise hidden malicious behavior. The framework uses an extensible set of value providers to fuzz the return values of APIs used by the app to interact with its environment. Given a set of target locations, an evolutionary algorithm-based search strategy steers the app to a target by combining suitable environment values. Our evaluation with 209 real-world malware apps shows that FuzzDroid effectively and efficiently reaches target locations and that it outperforms the closest existing approach.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) and by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, as well as by the German Research Foundation (DFG) within ConcSys.

REFERENCES

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. Grodddroid: a gorilla for triggering malicious behaviors. In *MALWARE*. IEEE, 2015.

[2] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[3] Android Security Team. Android security 2015 year in review, April 2016.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, June 2014.

[5] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 1–6. ACM, 2015.

[6] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In *Proceedings of the 26th International Conference On Automated Software Engineering (ASE '11)*, pages 53–62, Lawrence, KS, USA, 6-10 November 2011. IEEE.

[7] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX, 2008.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[11] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *WCRE '09*, pages 167–176, Washington, DC, USA, 2009. IEEE Computer Society.

[12] L. de Moura and N. Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[13] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[14] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *ICST'12: Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[15] Y. Fratantonio, A. Bianchi, W. K. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*.

[16] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[19] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[20] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[21] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77. ACM, 2013.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] C. Kolbitsch, B. Livshits, B. G. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, pages 443–457, 2012.

[24] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

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

[26] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

[27] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[28] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[29] D. Nakajima. Vietnamese adult apps on google play open gate to sms trojans. McAfee Labs Website, Jan 2014. http://blogs.mcafee.com/mcafee-labs/vietnamese-adult-apps-google-play-open-gate-to-sms-trojan.

[30] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013. USENIX.

[31] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*, 2014.

[32] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[33] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. How current android malware seeks to evade automated code analysis. In *9th International Conference on Information Security Theory and Practice (WISTP'2015)*, 2015.

[34] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[35] R. Sasnauskas and J. Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014, San Jose, CA, USA, July 22, 2014*, pages 1–5, 2014.

[36] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[37] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM, 2005.

[38] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA*, pages 189–206, 2011.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135. IBM, 1999.

[40] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, 2016.

[41] Z. Xu, J. Zhang, G. Gu, and Z. Lin. Goldeneye: Efficiently and effectively unveiling malwares targeted environment. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 22–45. Springer International Publishing, 2014.

[42] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, 2012. USENIX.

[43] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, page 68, 2013.

[44] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.