IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware

Michelle Y. Wong and David Lie
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—While dynamic malware analysis methods generally provide better precision than purely static methods, they have the key drawback that they can only detect malicious behavior if it is executed during analysis. This requires inputs that trigger the malicious behavior to be applied during execution. All current methods, such as hard-coded tests, random fuzzing and concolic testing, can provide good coverage but are inefficient because they are unaware of the specific capabilities of the dynamic analysis tool. In this work, we introduce IntelliDroid, a generic Android input generator that can be configured to produce inputs specific to a dynamic analysis tool, for the analysis of any Android application. Furthermore, IntelliDroid is capable of determining the precise order that the inputs must be injected, and injects them at what we call the device-framework interface such that system fidelity is preserved. This enables it to be paired with full-system dynamic analysis tools such as TaintDroid. Our experiments demonstrate that IntelliDroid requires an average of 72 inputs and only needs to execute an average of 5% of the application to detect malicious behavior. When evaluated on 75 instances of malicious behavior, IntelliDroid successfully identifies the behavior, extracts path constraints, and executes the malicious code in all but 5 cases. On average, IntelliDroid performs these tasks in 138.4 seconds per application.**

I. Introduction

Smartphone malware is and will continue to be a major security threat. One of the most attractive features of a smartphone is the ability to extend its functionality with third-party applications. Unfortunately, such a feature inevitably brings with it the threat of malicious smartphone applications, otherwise known as *smartphone malware*. With a reported 1.3 billion smartphones sold in 2014 [20], this large population of potential victims gives malware writers ample motivation to target smartphone devices, as indicated by a recent report by Sophos, which states that the number of new smartphone malware samples detected has doubled from 1000 per day in 2013 to 2000 per day in 2014 [38].

To combat the spread of malware, many application markets, such as Google Play and the Apple App Store, spend considerable resources trying to detect and remove malware. In addition to manual analysis by humans, these markets also employ automated techniques, such as static analysis, to identify applications that are likely to be malicious. However, static analysis techniques are inherently imprecise as they can only operate on an abstraction of the program [21]. As a result, such markets also employ dynamic analysis, which gathers information by executing the application. Unfortunately, while dynamic analysis is precise, it can only detect malicious behavior if the code that implements that behavior is executed during the analysis.

There are several common strategies that can be employed to trigger the malicious behavior that the dynamic analysis is trying to detect. The simplest, but least effective, is to have a predefined script of common inputs that will be executed on the application under analysis. This not only has a very low chance of triggering the malicious behavior, but can be easily evaded by a knowledgeable adversary. A more sophisticated approach is random fuzzing [31], [46], which applies randomly generated inputs on the application in an effort to trigger as many behaviors as possible. However, random fuzzing is inefficient, as it may generate many inputs that trigger the same code and behavior in the application. To address this, a recent and more effective technique is concolic testing [26], [2], which uses symbolically derived path constraints to exercise different paths in the application for each generated input.

However, none of these methods are ideal for triggering malicious behavior in applications because they are blind to distinguishing between code that performs the behavior that the dynamic analysis is trying to detect and code that does not. Thus, while concolic testing can efficiently achieve high code coverage, it will still waste many compute cycles by having the dynamic analysis analyze irrelevant parts of the application. Instead, we propose *targeted analysis*, which uses information about the dynamic tool in combination with static analysis of the application to generate a reasonably small set of inputs that will trigger the malicious behavior to be detected by the dynamic analysis. We implement and evaluate the concept of targeted analysis for detecting Android malware in a prototype we call *IntelliDroid*.

Naturally, it would be very difficult for a static analysis tool to generate only the exact inputs that are needed to trigger malicious activity in an application, as this would imply that the static analysis tool is as precise as the dynamic tool at detecting malicious behavior. Instead, we need a reasonably accurate over-approximation of the behaviors that will be analyzed by the dynamic tool. IntelliDroid can then generate a small set of inputs that will trigger all of the code matching the over-

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23118

approximation and allow the dynamic analysis to decide if it is actually malicious or not. For this approximation, IntelliDroid uses a list of "targeted" Android APIs that is specific to the dynamic analysis tool. This design decision is motivated by the observation that most malicious behaviors, such as sending and intercepting SMS messages, leaking private information or making malicious network requests, require the use of APIs [49][1]. In addition, malware may obfuscate malicious activity using reflection or dynamic class loading. IntelliDroid can trigger the reflection and class loading APIs so that the dynamic tool can observe the resolution of the reflected calls or the behavior of the dynamically loaded classes.

It is crucial that IntelliDroid can generate inputs that trigger all targeted APIs. Most tools that combine static and dynamic techniques use the dynamic analysis to prune false positives generated by the static analysis [28], [24]; thus, if the dynamic phase is unable to execute a particular path, it only increases the number of false positives. However, if IntelliDroid fails to trigger a malicious behavior, this will result a false negative, with more serious security consequences.

IntelliDroid introduces two new input generation and injection techniques that enable it to trigger code paths on which previous Android input generation techniques would fail [28], [24], [45], [48], [37], [9]. First, Android applications do not have a single entry-point, but are instead composed of a collection of event handlers. It can be insufficient to call just the event handler that contains a particular API invocation. Instead, the event handlers need to be triggered in a particular order, and in some cases a "chain" of several handlers needs to be triggered with specific inputs. IntelliDroid iteratively detects such *event-chains* and computes the appropriate inputs to inject, as well as the order in which to inject them.

Second, while previous work injects inputs at the application boundary [28], [40], [45], [35], this low-fidelity injection can lead to false application behavior because the application state is inconsistent with the Android system state. For example, to hide the presence of SMS messages from the user, an Android malware program could register an event handler for an incoming SMS, and then access and search the SMS content provider to delete the received message. Simply injecting the SMS notification at the application boundary will result in inconsistent behavior because the application expects the message to be in the SMS content provider database, but the Android framework itself has received no such message. IntelliDroid maintains environment consistency after input injection by injecting inputs at the lower-level *device-framework interface*, allowing all state in the Android framework to be automatically changed consistently. This high-fidelity input injection means that IntelliDroid can be integrated with essentially any dynamic analysis tool, including full system analysis tools such as TaintDroid [19].

In summary, we make three main contributions in this paper:

1) We present the design and implementation of IntelliDroid, an input generator that takes into account the malicious behavior a dynamic analysis tool is trying to detect. IntelliDroid uses targeted APIs as an over-approximation for these malicious behaviors and generates inputs that trigger all instances of those APIs in an application. We describe two novel techniques that enable IntelliDroid to trigger targeted APIs with injected inputs: detecting event-chains and device-framework interface input injection.
2) We show that the targeted API abstraction makes IntelliDroid easy to use with a dynamic analysis tool by integrating it with the TaintDroid dynamic analysis tool. When run on a corpus of malware, we show that IntelliDroid using TaintDroid can trigger and detect all privacy leaks. We also show that IntelliDroid's event-chain detection and device-framework interface input injection enable it to effectively generate inputs that trigger 70/75 targeted APIs in a corpus of malware.
3) We show that IntelliDroid is cheap and fast, requiring only 138.4 seconds of analysis time on average to successfully generate inputs to trigger targeted APIs on a corpus of malicious and benign applications. We also show that IntelliDroid is able to avoid running approximately 95% of an application during dynamic analysis while still detecting all malicious behaviors.

We begin by giving relevant background on Android and static analysis in Section II. We then describe the design of IntelliDroid in Section III. Details about the implementation are in Section IV. Evaluation showing the effectiveness and performance of IntelliDroid are presented in Section V. Section VI goes over the limitations, while related works are discussed in Section VII. Finally, we conclude in Section VIII.

II. Background

Though documentation about the Android programming environment and system are widely available[2], a perhaps less well-documented aspect of Android is the implementation of the *Android framework*, which forms the middle layer between third-party applications and Android's custom Linux kernel. The Android framework consists of system services that communicate with the device's hardware components, as well as classes that implement application programming interface (API) methods that third-party applications invoke. When an application is launched, execution begins in the Android framework, which loads the application components and manages their lifecycles. Applications on the Android platform are event-driven and implement specific methods, which we call *entry-points*, that are invoked by the framework when events, such as location events from the GPS sensor or SMS events from the cellular chip, occur. Some application entry-points must be registered with the framework, which will then invoke them when the associated external event occurs.

Together with the APIs that applications can invoke, these entry-point methods form the *framework-application interface* that divides the Android framework from code that originates from third-party applications. At the other end is the interface between the framework and the underlying devices from which external events are generated, which we call the *device-framework interface*. When a device sensor receives new data, it notifies the framework so that the event data can be processed

[1]Our own analysis shows this is true of more recent malware as well as the older malware in the cited study.

[2]Please see http://developer.android.com/index.html

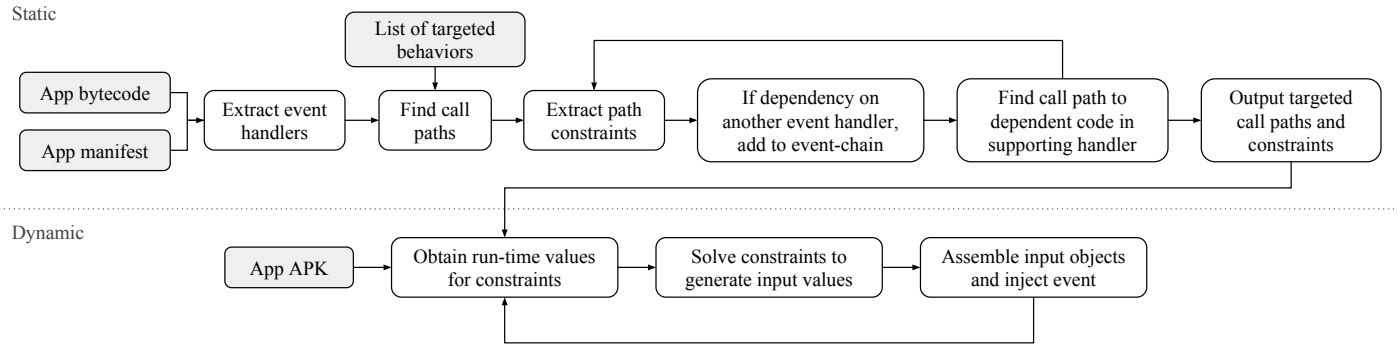


Fig. 1. IntelliDroid System Diagram

and disseminated to applications by calling the corresponding entry-points in the framework-application interface.

In addition to invoking application event handlers, system services within the framework also store information about the event as it is processed. This allows applications to refer to the event and obtain extra information at a later time. Some services, such as SMS, store all past event information in a content provider to be queried and modified by applications with sufficient permissions. Other services, such as location, store only the last event received. In both cases, the handler invocation in the application and the event information stored in the framework must be kept consistent for correct execution.

III. Design

IntelliDroid generates inputs for a dynamic analysis tool that monitors the execution of an Android application. Given a set of targeted APIs that represent the analysis performed by the dynamic tool, IntelliDroid will find instances of these targeted APIs in the application and generate inputs to trigger them. These inputs can be injected into an actual Android system, allowing IntelliDroid to be integrated with any dynamic analysis tool, including those that monitor application execution from an instrumented OS [19], or from a virtual machine emulator [39]. To accomplish this task, IntelliDroid takes the following steps:

1) Identify invocations of targeted APIs. For each invocation, identify the *event handlers* where execution begins in the application code and find *target call paths* from the handlers that lead to the targeted API.
2) For each call path, extract path constraints that control whether the targeted API is invoked or not.
3) In cases where the constraints depend on execution paths in other event handlers, extract the necessary constraints and the order of the *event-chain* that is required to invoke the targeted API.
4) Using an off-the-shelf constraint solver, solve the path constraints to determine the necessary input values that will lead to invocation of the targeted API. Some inputs may depend on external values, such as responses to network requests, so IntelliDroid dynamically extracts the concrete values for these external dependencies during execution.
5) Apply the computed input values in the appropriate order to the *device-framework interface*. This will consistently execute the appropriate paths required for the targeted API to execute. IntelliDroid contains a modified Android OS that can inject inputs at the device-framework boundary.

The flow between these tasks in the IntelliDroid system is shown in Figure 1, which also differentiates between the static and dynamic components. Our current prototype uses the WALA [41] framework for static analysis and Z3 [17] as the constraint solver.

In addition to the above, IntelliDroid has the capability to both *monitor* and *control* interaction between the application and external components. For example, some malware instances will contact an external server to get a list of SMS numbers to intercept or send messages to. In these cases, it might be appropriate to monitor these interactions so that IntelliDroid can generate inputs matching these requirements. In other instances, we may need to simulate a message from a remote server to trigger a targeted API, in which case it might be appropriate to control that interaction.

A. Specifying Targeted APIs

Because we want IntelliDroid to be applicable to as wide a range of dynamic analysis tools as possible, we need to select a suitable abstraction that over-approximates the types of behavior that various dynamic analysis tools are trying to detect. As stated previously, an initial justification of our choice to use Android APIs for this abstraction is that most malicious behaviors require an application to invoke an Android API. As further justification, we perform a survey of recent Android malware dynamic analysis techniques that have been proposed in the literature. From our results in Table I, we show that dynamic analysis tools that detect Android malware can be separated into three categories depending on their operation: (1) by analyzing invocations to certain API methods; (2) by analyzing invocations to system calls; and (3) by analyzing low-level side effects of the application, such as CPU load or battery usage. We find that specifying behaviors as API methods allows IntelliDroid to cover most of the current dynamic tools. We elaborate on our reasoning below.

1) Analyzing API Methods: The vast majority of dynamic analysis tools analyze API method invocations and the target methods for IntelliDroid can be determined by analyzing the specific API methods used to configure the tool. For instance, TaintDroid [19] performs taint tracking by adding taint tags in locations where sensitive information is obtained by the application (i.e., sources) and reading taint tags in locations where

TABLE I. EXISTING ANDROID DYNAMIC ANALYSIS TOOLS

Dynamic Tool	Goal	Features for Analysis
AASandbox [10]	Monitor behavior via tracking of system calls	System calls
Andromaly [36]	Malware detection via system resource usage	Low-level device features (e.g. battery usage, CPU load)
CopperDroid [39]	Monitor behavior via system call tracking	System calls
Crowdroid [12]	Monitor behavior via tracking of system calls	System calls
DroidBox [18]	Sandbox to monitor external accesses	Sink API methods
DroidRanger [50]	Detect malware using prespecified behavioral footprints and heuristics	Sequence of API method invocations and parameters
DroidScope [39]	Plugins for API tracking, instruction tracing, and taint tracking	API methods; source/sink API methods
RiskRanker [39]	Detect malware using known vulnerability signatures	Sequence of API method invocations
TaintDroid [19]	Detect privacy leakage	Source/sink API methods
VetDroid [47]	Malware detection via permission use behavior	Permission requests (can be mapped to API methods)

information leaves the application (i.e., sinks). By referring to the documentation or searching through the source code, these methods can be found and used as target methods. Sandboxing tools such as DroidBox [18] track locations where data leaves the application and target methods can be determined by finding the instrumented API methods. Other tools such as DroidScope [44] allow the user to trace specific API method invocations; these API methods would serve as target methods.

Some dynamic tools, such as VetDroid [47], detect malware by dynamically analyzing an application's permission usage. Although the tool does not trace API methods, the mapping between permission use and API methods has been well-studied and can be obtained from PScout [5] or Stowaway [22]. IntelliDroid can therefore be configured with target methods that map to the permissions of interest. Since the majority of dynamic analysis tools analyze API calls, using API calls as our abstraction would enables IntelliDroid to generate inputs for most of the dynamic analysis tools.

2) Analyzing System Calls: The next most common method used by dynamic analysis tools is to analyze system calls. In this case, the user must determine a mapping between the system call method and API methods that use the system call. Such tools include CopperDroid [39], AASandbox [10] and Crowdroid [12]. If only specific system calls are traced (e.g. file access), the user can use Android's documentation to find API methods that use the system call's functionality and generate the mapping manually. In general, however, it can be difficult to map every system call to API methods in this manner; therefore, the user may need to perform a one-time static analysis of the Android framework. A backward traversal of the framework's call graph from the invocations of system calls to public API methods should provide the necessary mapping, which can then be used to obtain the target methods. As a result, we believe IntelliDroid would be able to generate inputs for dynamic analysis tools that analyze system calls, albeit with more effort required on the part of the user.

3) Analyzing Low-Level Events: A few tools focus on analyzing low-level events on the device. Andromaly [36] is one such dynamic tool that tries to infer malicious activity by detecting anomalies in CPU load and battery usage during the application's execution. The ability to attach IntelliDroid to such analysis tools depends on how the features are being traced. If the tool merely detects single instances of usage, it may be possible to use IntelliDroid to trigger API methods that correspond to those resources, such as those that invoke the camera or GPS. However, IntelliDroid is not an appropriate input generator for analysis tools that profile anomalies in resource usage over time, as the IntelliDroid does not seek to mimic realistic usage. In such cases, it would be more effective to use a tool that aims to replicate normal use or have a user manually execute the application.

While the specification of the targeted APIs for a dynamic analysis is manual, it is not overly onerous. We demonstrate this by extracting the APIs and having IntelliDroid generate inputs for TaintDroid, for which we further discuss the associated effort and effectiveness in Section V.

Although the IntelliDroid currently uses Android APIs to represent behaviors that the dynamic analysis targets, the design allows other forms of targets to be specified. In general, if the user can determine a point in the code to which execution is desired, this information can be given to IntelliDroid, which will extract the call paths and path constraints to the specified code location. This location can be as simple as a method invocation, or can be derived from some other analysis. For instance, to direct execution for a dynamic tool that focuses on native code usage, IntelliDroid can be configured to extract paths and constraints for invocations to native methods.

B. Identifying Paths to Targeted APIs

For a given application and set of targeted APIs, IntelliDroid first performs static analysis to identify the invocations of targeted APIs and the paths leading to them. Because Android is event-driven, an application may contain several entry-point methods where the Android framework can transfer execution to the application. These methods are normally event handlers that receive various system events, such as callbacks to control a component's lifecycle, process sensor inputs, or respond to UI events. Using an entry-point discovery mechanism similar to FlowDroid's [4], the application's components are read from its manifest and their lifecycle methods are extracted. A partial call graph created from these entry-points is used to search for instantiations of Android callback listeners and to add overridden listener methods to the list of entry-points. A new partial call graph is generated and the process is repeated iteratively until no more entry-points are found.

The call graphs are generated using the event handler entry-points as starting points for the code traversal. However, Android mechanisms such as Intents and asynchronous calls can cause execution to flow between methods in an application even when there is no explicit function call. For instance, Android allows execution to be transferred between different components of an application using the `Intent` interface, so the points at which Android intents are sent and received are

```
1  class SmsReceiver extends BroadcastReceiver {
2    String sNum;
3
4    void onReceive(Context c, Intent i) {
5      if (i.getAction()=="SMS_RECEIVED") {
6        handleSms(i);
7      } else if (i.getAction()=="BOOT_COMPLETED"){
8        this.sNum = "99765";
9      }
10   }
11   void handleSms(Intent i) {
12     Bundle b = i.getExtras();
13     Object[] pdus = (Object[])b.get("pdus");
14
15     for (int x = 0; x < pdus.length; x++) {
16       SmsMessage msg =
             SmsMessage.createFromPdu(pdus[x]);
17       String addr = msg.getOriginatingAddress();
18       String body = msg.getMessageBody();
19       // Constraint depends on local function
20       if (needsReply(addr, body)) {
21         SmsManager sm = SmsManager.getDefault();
22         sm.sendTextMessage(addr, null, "YES", null,
               null);
23       }
24       // Constraint depends on heap variable
25       if (addr.equals(this.sNum)) {
26         abortBroadcast();
27       }
28     }
29   }
30   boolean needsReply(String addr, String body) {
31     if ((addr.startsWith("10658") &&
32          body.contains("RESPOND")) ||
33         (addr.startsWith("10086") &&
34          body.contains("REPLY"))) {
35       return true;
36     }
37     return false;
38   }
39 }
```

Listing 1. Code Example

identified and the execution flow between them are represented by additional edges in the call graph. Details regarding other Android-specific call edges can be found in Section IV-A.

A traversal for each event handler is then performed to identify the event handlers that may lead to a targeted API invocation. For each targeted API, a *target call path* is extracted, which contains the sequence of method invocations from the event handler entry-point to the invocation of the targeted API.

To illustrate the design of IntelliDroid, we use the code provided in Listing 1 as an example throughout this section. This code is derived from several malicious applications, and is representative of malware that intercepts and automatically responds to SMS messages received from a malicious party using the Android APIs `BroadcastReceiver.abortBroadcast` and `SmsManager.sendTextMessage`, respectively. We will assume that the malware will be analyzed with a dynamic analysis tool that is capable of detecting malicious SMS usage and thus, we specify all SMS-related Android APIs (including `sendTextMessage` and `abortBroadcast`) as targeted APIs to IntelliDroid. IntelliDroid will begin analyzing the example by identifying `SmsReceiver.onReceive` at line 4 as an event handler entry-point and the calls to `sendTextMessage` and `abortBroadcast` at lines 22 and 26 as targeted APIs. IntelliDroid then identifies the target call paths from the event handler to each of the targeted APIs, which are the paths through the method invocations on lines 4→11→22 and 4→11→26.

C. Extracting Call Path Constraints

To actually trigger a target call path, the appropriate inputs must be injected into the application. For each method in the target call path, the invocation of the next method in the path may be control-dependent on conditional branches in the method body. To extract these control dependencies, a forward control- and data-flow analysis is performed on the control flow graph (CFG) of each method. The control-flow analysis determines whether a conditional branch affects execution of the next method's invocation, and if so, it extracts constraints based on the variables used in the predicate of the branch statement. IntelliDroid also extracts symbolic data-flow information about variables to identify other variables they may depend on, or that may depend on it. Similar to traditional data-flow analyses, loop support is implemented by performing the analysis iteratively until the constraint and data-flow output converges.

The forward control-flow and data-flow analysis is obtained by propagating variable information along the method's CFG, both locally within each basic block and across different basic blocks. Constraints are generated by translating the operator and variables used by each instruction encountered. If multiple CFG paths are found to lead to the next method invocation, the analysis combines the extracted constraints of each path with a logical OR ($\vee$), indicating that as long as one path is satisfied, the targeted API is executable. The extracted constraints for each path method are combined using full context-sensitivity, which is achievable since the call site of each method along the path is known.

As an example, consider the execution path in Listing 1 ending in the `abortBroadcast` invocation at line 26. The target call path generated by IntelliDroid includes an invocation of `handleSms` in the `onReceive` method, which is dependent on the intent action string; therefore, the constraint (`i.getAction() = "SMS_RECEIVED"`) would be extracted. IntelliDroid's context-sensitive inter-procedural analysis would indicate that when `handleSms` is called, the invocation parameter originated from the event handler's input parameter (`Intent i`). Analysis through `handleSms` shows that the execution of the target API invocation is dependent on the length of the PDU array and the originating address of the SMS message received. The conditional statement in line 20 is on the execution path, but since the targeted API can be reached regardless of the branch outcome, it has no effect — when processing the control flow, the constraints extracted from both sides of this branch would be combined with the OR operator.

In some cases, the variables extracted for the constraints are return values from other method invocations. Although these methods are not part of the target call path, their return values affect the execution of the path and the constraints they impose must be extracted. An example of such an *auxiliary method* is `needsReply` in Listing 1. To handle such cases, IntelliDroid extracts constraints for the return values and the paths leading to the return sites within the auxiliary method. These auxiliary constraints are combined with the main path constraints with a

logical AND ($\wedge$) to enforce a specific return value and return path through the auxiliary method.

For some situations, IntelliDroid also inserts library constraints manually extracted from Android API calls to pure functions — i.e., functions whose result depends only on their arguments, with no side-effects. For example, `addr.equals()` on line 25 is an invocation to a pure function and IntelliDroid will convert this to the constraint `(addr = this.sNum)`. In some cases, the API method invoked would generate constraints that are too large or complex for the constraint solver; this is the case for `createFromPdu` on line 16, which performs bytewise operations on the bytecode of the SMS message. In these cases, rather than rely on the constraint solver, we provide IntelliDroid with a manually implemented function that inverts `createFromPdu`, thus allowing IntelliDroid to generate an appropriate input. This is conceptually equivalent to "stitching", which is used to solve constraints that contain similarly complex functions, such as SHA1 and MD5, in BitFuzz [13]. Android API methods that are not pure functions must be handled dynamically at run-time by either monitoring or controlling them, as described below in Section III-E.

D. Extracting Event-Chains

At this point, the extracted constraints consist of a boolean expression of concrete values and variables. Ideally, all of the constraint variables should be dependent on the event handler's input parameters. In such a case, solving the constraints for these variables and injecting the solved values will execute the desired target call path. However, there may be cases where the path constraints depend on heap variables that cannot be set to the correct values using only the arguments to the entry-point method of the target path. In this case, IntelliDroid must find a definition for these variables that can be executed to set the heap variable to the required value.

An example of this is shown in Listing 1 for the `SmsReceiver.sNum` heap variable, in the call path to `abortBroadcast`. This variable is used in a constraint imposed by the conditional branch in line 25, but is defined in another invocation of `onReceive` where the intent action string is `BOOT_COMPLETED`. To complete the constraints and execute the target path, two actions must be completed: (i) IntelliDroid must find any additional constraints on the heap variable and add them to the current path constraints; and (ii) IntelliDroid must ensure that the value extracted for the target constraints is actually stored in the heap variable prior to executing the target path. Thus, when the constraints contain a heap dependence, IntelliDroid searches for statements where the heap variable is defined and records the event handlers containing the definitions. The path from the event handler to the store instruction becomes a *supporting call path* and IntelliDroid extracts *supporting constraints* for this path in the same manner used for the target path. Later, when solving the constraints, a concrete value will be assigned to the heap variable and used to solve the target path constraint.

For `sNum` in Listing 1, the supporting call path would begin at `onReceive` and the supporting constraints would include `(i.getAction() = BOOT_COMPLETED)`. The main target path constraints would be appended with the extra constraint (`sNum = 99765`), which is extracted from the supporting path and ensures that the SMS originating address is properly constrained. The supporting path and the main target path form an *event-chain* that results in the activation of the targeted API. In the run-time component of IntelliDroid, this event-chain will result in multiple input injections. In cases of multiple heap variable dependencies, the process is performed iteratively, as shown by the topmost backedge in Figure 1. This forms a event-chain ordered by the data-flow dependency between the variables.

Event-chains are also needed when event handlers are registered with the Android framework. In the Android system, some event handlers are known to the system (e.g. lifecycle handlers), some are declared in the application's manifest, and some are registered dynamically within the execution path of a previous event handler. For those that are registered dynamically, the registration process may require parameters specifying how and when the event handler is to be called. For instance, registering location callbacks requires that the application specify the frequency and minimum distance between consecutive callback invocations. These values are added to the constraints to ensure that the injected event abides by these parameters in the same way the Android framework would in normal execution. The supporting call path leading to the event handler registration is added to the event-chain due to the control-flow dependency between it and the target call path.

E. Run-Time Constraints

For the simple case where all constraint variables are input-dependent or can be concretized, the constraints can be solved statically and the run-time system merely has to inject the input values. However, there may be cases where the values of the non-input-dependent constraint variables cannot be determined statically. This may occur for heap variables where the alias analysis is imprecise or for values obtained from Android API methods that cannot be modeled statically. A purely static constraint extraction approach would either be unable to extract all the path constraints or would need a considerably more precise and expensive static analysis to do so. However, IntelliDroid's hybrid static-dynamic design sidesteps this dilemma by obtaining the values during run-time by performing the constraint solving step immediately prior to event injection. That is, although the constraints are extracted during static analysis, they are solved at run-time so that any statically unresolved variables can be resolved prior to being passed to the constraint solver.

This delayed solving gives the user a choice of how to use IntelliDroid to handle interactions between the Android application and its environment. Variables whose values depend on these interactions can either be *monitored*, indicating that the interaction is allowed to proceed without modification and IntelliDroid merely eavesdrops on the interaction, or *controlled*, where the IntelliDroid intercepts and replaces these interactions with values it determines will exercise interesting paths. Variables that depend on input from a possibly malicious external component can be either monitored to understand the interaction of the malware with the external component, or controlled to understand what potential capabilities the malware may give to an adversary in control of the external component. On the other hand, variables that are derived from

the Android framework and OS, which are trusted, would generally be controlled to take a value that, together with the other constrained variables, satisfies the target path constraints and enables the target API to be executed.

For monitored variables, the external input is often derived from a control server that sends commands to the application. In some cases, the application may request data from the server and use this data to perform malicious activities. A common example of this presented in [49] involves applications that download a list of premium SMS numbers from the network and intercept messages to/from these numbers such that the user is unaware of the premium fees. Although the server input cannot be determined statically, network monitoring can extract the values returned and add these values to the target path constraints. Constraints that occur on the server side are not captured by IntelliDroid, although it is possible to set up a fake server that sends the necessary replies to the application when it makes network requests. However, because the fake replies can affect the malicious activity that IntelliDroid aims to analyze, these external variables are instead monitored to determine the real values that the application expects and how it behaves when given these values.

For controlled variables, they are unresolved due to their dependence on the device state. For instance, malicious behavior may only manifest during a certain time or date, and this is reflected by constraints that contain the system time/date as variables. These variables can be resolved by setting the device state (e.g. setting the time) prior to injecting the main event. The actual value used is determined by the statically extracted constraints that depend on the external variable. For instance, if a constraint is extracted stating that a particular target API invocation is only triggered only when the system time is set to "1:00", the device time will be set to this value before injecting the inputs to trigger the target API. This is essentially another form of event-chain extraction, where supporting events must be injected prior to executing the main target call path.

F. Input Injection to Trigger Call Path

Once the constraints are generated and all run-time values obtained, IntelliDroid can trigger the desired target call path by obtaining the input parameters that fulfill the constraints. As previously discussed, the task of solving the constraints is placed in the dynamic component of IntelliDroid; thus, the input parameters are solved for and generated immediately prior to executing the target call path.

The dynamic component of IntelliDroid consists of a client program running on a computer attached to the device. Communication between this program and the device is facilitated by a newly constructed Android service (`IntelliDroidService`) that serves as a gateway for the tool. As motivated earlier, IntelliDroid must inject inputs at the device-framework interface, rather than the application event handler, to ensure that state in the Android framework is consistent with application state. When the static component specifies inputs for the execution of the targeted API, the gateway service is responsible for injecting that input into the device-framework interface of the Android OS on which the application is running. To do this, `IntelliDroidService` must perform two tasks. First, it must identify a suitable injection point. Second, it must format the input values for injection into the Android OS.

To identify a suitable injection point, IntelliDroid must identify a method at the device-framework interface that: (1) is called when the corresponding external event occurs; and (2) directly calls the desired application handler when it is invoked. Further, such input injection points must have a one-to-one relationship with the event handler of interest, so that inputs thus injected will only result in the invocation of the desired application event handler and no other handlers. For instance, SMS events are received by the framework via a socket, which is monitored by a long-running process. When an SMS message arrives on the socket, a device-framework interface is invoked by the process, which eventually calls `PhoneBase.sendMessage`, the desired device-framework interface handler.

To find suitable injection points, we perform static analysis of the Android framework, using a backward call graph traversal starting from the event handlers of interest to find candidate injection points. Alternatively, since these injection points are often located in Android service classes and these service classes are well-known, IntelliDroid can be given a list of classes where injection should occur and it will automatically generate paths between methods within these classes and the event handlers to be triggered. Because invoking such injection methods will often require interprocess communication, IntelliDroid preferentially selects RPC methods as input injection points as they present a cleaner interface.

To properly format inputs values for injection, the input constraints for the application event handler (extracted by the static phase of IntelliDroid) must be transformed into constraints at the input injection point and then solved. As a result, constraints imposed on the *injection path* between the injection method and the application event handler are extracted using the same analysis that IntelliDroid performs on applications. In some cases, injection paths may have dependencies on other paths in the framework, requiring a chain of device-framework events to be injected to properly invoke the application event handler.

Since the Android framework is the same for every application, IntelliDroid extracts the injection points and injection path constraints for supported application event handlers once and stores them in a library for use at run-time. At run-time, injection path constraints are combined with target call path constraints in the application using a logical AND. In addition, IntelliDroid appends extra constraints specifying how the injection method parameters are related to the event handler parameters. Finally, the inputs may need to be formatted by initializing the fields of a specific input object (for instance, a `Location` object for a location event) to the desired value. While the constraint solver can automatically generate the appropriate values for the fields, the code to populate them in the object is manually implemented.

IV. Implementation Details

A. Static Analysis

For versatility, IntelliDroid performs its analysis on compiled Android applications and does not require source code.

Because they are packaged in APK files and stored as Dex bytecode, the applications must be unpacked and converted to Java bytecode prior to analysis, using tools such as Dare [33] and APKParser [3]. The converted files are then passed to IntelliDroid's static component, which uses the WALA static analysis libraries [41]. WALA provides support for basic static analysis, such as call graph generation, data flow analysis, alias analysis, and an intermediate representation based on SSA.

To perform the actual analysis on the code, IntelliDroid creates a call graph using 0-1 context sensitivity with a type-based heap model; this call graph is used to search for the targeted APIs. However, the Android platform provides facilities, such as `Intents`, `Threads`, `Executors`, IPCs, RPCs and `AsyncTask` to allow applications to transfer execution between event handlers without an explicit method invocation. When generating the call graph within WALA, these Android-specific edges are automatically added such that the call graph can give an accurate representation of how execution flows between methods in the application. The call edges are conservatively patched based on the documented behavior of the invoked method and on the parameters or constant values used in the invocation. While a less precise call graph is used when searching for the target methods, more precision is added when analyzing individual target call paths to resolve method invocations with full context and perform pointer analysis for heap dependencies.

There are certain cases where framework API method invocations must be treated differently. For instance, when the constraint extraction encounters API methods that obtain information from external sources (such as the network or a file), it must note whether the returned values can be controlled or monitored. This distinction is currently made on a per-method case and is determined by whether the source of the data is controlled by the third-party application developer. Any data originating from an external source other than the device, Android framework, or Android OS is considered potentially malicious and the value is monitored. Other framework methods may also be modeled due to the limitations of the constraint solver. For instance, string methods are modeled internally as well as trigonometric operations, since the constraint solver does not support such functionality. In general, processing the invocation of a framework method depends on whether it introduces externally-obtained data and whether the constraint solver supports the operations performed.

The constraints generated by IntelliDroid are placed into an app-specific file. When the static phase has completed, this file will contain all target call paths found in the application, along with information detailing how the dynamic component can trigger them. For a given application, only one execution of the static component is needed, since this file will contain all of the information that the dynamic component requires.

B. Dynamic Analysis

The dynamic component of IntelliDroid consists of a client program running on a computer, connected to a device or emulator with a custom version of Android. The dynamic client program is implemented using Python and acts as the controller that determines the target call path to execute. It also interfaces with the constraint solver used to generate the path inputs: the Z3 constraint solver [17] with the Python API (Z3-py). Communication between this program and the device is facilitated via sockets, using the device port-forwarding feature of the Android Debug Bridge (ADB) [3]. The other endpoint of the socket is located in the gateway Android service, `IntelliDroidService`. The `IntelliDroidService` class is implemented as a long-running system service that is instantiated upon device boot. On receipt of messages from the client program, this service can obtain information about event handlers, assemble an input object using values that the client program sends, and trigger an event with the input object.

In certain cases, run-time values for constraints in the injection path are needed. For instance, the `onLocationChange` event handler is called only when there is a minimum distance from the last location sent to the application. The constraint modeling this relationship would require the value of the last location that the event handler received, as well as the minimum distance parameter stored in the framework. IntelliDroid extracts these values during run-time, by instrumenting the system services handling these events to send event handler information when requested by `IntelliDroidService`. Although such run-time extraction is not strictly necessary, it can provide an advantage over static extraction in cases where the event handler registration parameters are not explicit within the application code.

Because IntelliDroid is currently using the Python API for the Z3 solver [17], the Z3 string library is not available. Therefore, string functions such as `equals()`, `contains()`, or `startsWith()` must be modeled and string variable types are handled by the dynamic component as a special case. Due to the heuristics used when modeling such functions, there can be cases where complex string manipulation may not be represented precisely by the extracted constraints.

V. Evaluation

Our IntelliDroid prototype is implemented for the Android 4.3 operating system (Ice Cream Sandwich) and evaluated on an Intel i7-2600 (Sandy Bridge) CPU at 3.40 GHz with 16GB of memory. In the evaluation, we aim to answer the following questions:

- *How effective is targeting API calls derived from a real dynamic analysis tool, and can this technique trigger all of the malicious behavior that the dynamic analysis tool can detect?*

  We integrate IntelliDroid with TaintDroid [19], a dynamic taint-tracking tool and we demonstrate that the combination is able to detect all sensitive data leaks in a corpus of privacy infringing malware.

- *Given a targeted API, how effective is IntelliDroid at generating the inputs to trigger it?*

  We test IntelliDroid on a wider range of targeted APIs and malware, and evaluate whether it can generate inputs to trigger all malicious behavior. We also discuss the effectiveness of the different techniques used by IntelliDroid, such as event chains and run-time data gathering.

[3] http://developer.android.com/tools/help/adb.html

TABLE II. TAINTDROID TARGETED APIS

API Type	Number of APIs
Read phone data	4
Read database	13
Read location	7
Read UI data	1
Read account data	1
Read media data	13
Write data to HTTP	8
Write data to SMS	4
Write data to file	11
Total	62

- *What performance benefits can IntelliDroid's targeting potentially provide for a dynamic analysis tool? What are the run-time costs of the static and dynamic components?*

  We measure the time IntelliDroid takes to generate and inject inputs, the number of inputs required, and the amount of code that IntelliDroid is able to avoid executing.

A. Targeted Execution with IntelliDroid-Targeted TaintDroid

To demonstrate how IntelliDroid can be used in practice, we integrated IntelliDroid with TaintDroid [19], a dynamic taint-tracking system, to produce a combined system we call Intelli-TaintDroid. Integration with TaintDroid is straightforward and requires the merging of IntelliDroid's input injection component with TaintDroid's code base, which can be done with an automated patch. To derive the set of targeted APIs from TaintDroid, we analyze TaintDroid's documentation and source code to identify the instrumented methods that add and check taint tags. In cases where taint is assigned or checked in an internal framework method, we traced the call path back to an API method. Table II summarizes the number and types of APIs targeted. We found that specifying the targeted APIs for TaintDroid was fairly easy and took the first author on the order of 2-3 hours to produce the full set of targeted APIs.

We perform three experiments with Intelli-TaintDroid. First, we evaluate against a malware set for which we know the ground truth of all malicious behaviors. In this way we can evaluate the accuracy of Intelli-TaintDroid. Second, we compare against FlowDroid, a purely static analysis tool that also detects privacy leakage. Finally, we compare against TaintDroid driven by Monkey, a generic non-targeted fuzzer.

To perform a ground-truth evaluation of Intelli-TaintDroid, we need malware for which all known privacy leaking behaviors are known. To this end, we use 14 documented malware families from the Android Malware Genome dataset [49] that are known to leak sensitive information and supplement this with several recent samples from the Contagio project [34], which we manually analyzed to find all privacy leaking behaviors. Table III summarizes all of the behaviors that the malware is known to exhibit. The Intelli-TaintDroid combination is able to detect all of these behaviors with no false positives. IntelliDroid generates the appropriate inputs that trigger the privacy leakages and TaintDroid's dynamic tracking promptly reports it. In some cases, tainted data may flow through the heap and this would require executing intermediate paths that do not directly invoke the targeted API methods. IntelliDroid's event-chain mechanism detects these flows and invokes the necessary intermediate events to complete the flow from taint source to taint sink.

TABLE III. PRIVACY LEAKING MALWARE

Malware	Leakage Paths	Sensitive Data
Backflash	SMS → SMS	SMS, IMEI
	SMS → HTTP	SMS, IMEI
	Lifecycle → HTTP	IMEI
	Boot → HTTP	IMEI
Bgserv	SMS → HTTP	phone number
	SMS → File	phone number
	SMS → HTTP	phone number
Cajino	Intent → HTTP	SMS, IMEI, contacts, files
CoinPirate	SMS → HTTP	SMS
Crusewin	SMS → HTTP	SMS
Endofday	SMS → File	phone number
GamblerSMS	SMS → SMS	SMS
GGTracker	SMS → HTTP	SMS, phone number
GoldDream	SMS → SMS	SMS
GPSSMSSpy	Location → SMS	location
NickyBot	SMS → SMS	IMEI
	Lifecycle → HTTP	IMEI
HeHe	SMS → HTTP	SMS, IMSI
	SMS → File	SMS
	Lifecycle → HTTP	IMEI, IMSI
NickySpy	Boot → SMS	IMEI
Pjapps	SMS → SMS	SMS
	Lifecycle → HTTP	IMEI
SMSReplicator	SMS → SMS	SMS
Spitmo	SMS → SMS	SMS
Zitmo	SMS → HTTP	SMS

We further compare Intelli-TaintDroid against FlowDroid [4], a purely static taint-tracking tool, on the same set of malware. Since FlowDroid uses a more sophisticated static analysis than IntelliDroid, we expect that it might be more complete than IntelliDroid. However, out of the 26 privacy leaks, FlowDroid is unable to precisely detect the leakage in 7 cases because it stops when the sensitive information is sent to an Intent. Since Intelli-TaintDroid executes the full system, it is able to detect that data sent to these intents is eventually leaked via SMS or HTTP. We also note that Intelli-TaintDroid has no false positives, though it does report extra leaks that FlowDroid does not, since TaintDroid also monitors system services while FlowDroid only analyzes the application. We manually confirmed these extra flows to be true privacy leaks.

To fully compare against FlowDroid, we also tested Intelli-TaintDroid with the DroidBench test suite used in FlowDroid's own evaluation. Although DroidBench was meant to evaluate static analysis tools, this comparison shows the advantages of dynamic analysis when attached to a targeted execution tool such as IntelliDroid. Intelli-TaintDroid is able to detect all privacy leaks without any of the false positives of FlowDroid, due to the increased precision of dynamic taint-tracking.

Finally, we compare our Intelli-TaintDroid implementation against TaintDroid on its own being driven by Monkey[4]. While Monkey is a simplistic tool, we found that we were unable

[4] http://developer.android.com/tools/help/monkey.html

TABLE IV. NUMBER OF INJECTED INPUTS REQUIRED BY INTELLIDROID TO TRIGGER MALICIOUS BEHAVIOR

Malware	Injections Required
Backflash	41
Bgserv	91
Cajino	167
CoinPirate	85
Crusewin	2
Endofday	44
GamblerSMS	5
GGTracker	9
GoldDream	43
GPSSMSSpy	19
HeHe	430
NickyBot	104
NickySpy	107
Pjapps	64
SMSReplicator	7
Spitmo	5
Zitmo	3
Average	72

to integrate more sophisticated open-source tools with TaintDroid. We had attempted to compare with DynoDroid [31] (only available on Android 2.3), but we were unable to integrate it with TaintDroid successfully. We were also similarly unsuccessful with integrating the Android concolic testing system ACTEve [2] with TaintDroid.

We ran Monkey on each application for one hour, sending over 60K injections per application. Since Monkey is only capable of sending UI events and select system events, Monkey-TaintDroid missed 21 out of 26 cases of privacy leaks in our malware dataset, where the leaks require non-UI events such as location or SMS. Monkey was also unable to trigger leaks in cases such as *GPSSPSSpy*, where specific input strings must be injected to trigger the privacy leak. In comparison, Table IV shows the number of input injections that Intelli-TaintDroid required to detect all malicious behavior in each application. We can see that overall, IntelliDroid needs between 2 and 430 inputs (with an average of 72) to trigger all malicious behavior in any one of our malware samples. While we speculate that DynoDroid would likely have been able to detect more leaks because it can inject non-UI events, we do not believe that it would be able to guess the correct input strings needed to trigger the privacy leak either. ACTEve, being a concolic testing tool that performs static analysis, would likely be able to determine the correct inputs, but as a coverage tool, it seeks to execute each path only once and thus may miss malicious behaviors since it does not know the order in which to inject inputs. In contrast, IntelliDroid injects each input once and determines from static analysis the correct order to inject them.

B. Generating Inputs to Trigger Targeted APIs

The previous section shows that IntelliDroid is effective in practice when integrated with a real dynamic analysis system. However, TaintDroid itself is only capable of detecting privacy leaks. We now seek to understand the limits of what types of inputs IntelliDroid can generate when tasked with triggering a larger variety of behaviors. To do this, we use 27 malware families from the Android Malware Genome [49] and Contagio datasets [34], and use a set of targeted APIs that would have been derived from a hypothetical tool that would be capable of detecting all known malicious behavior, given IntelliDroid's ability to trigger it. The malware in our dataset performs malicious actions that are typical of many types of malware, including SMS manipulation and monetization, receiving command and control messages via the network and SMS messages, sending stolen data over the network, and other malicious network requests. They also obfuscate their actions using techniques such as reflection and dynamic class loading, which are common among Java-based malware. In some cases, a malware sample can exhibit several malicious behaviors, giving the dataset a total of 75 malicious behaviors that IntelliDroid must trigger.

For each behavior, we describe both the targeted APIs that IntelliDroid targets (i.e., the static configuration), as well as how we confirm that IntelliDroid is able to successfully trigger the targeted API.

① **Premium SMS**: Trigger paths to `SmsManager.sendTextMessage`. Confirmed by checking that `sendTextMessage` is called with a premium number.

② **Blocking SMS**: Trigger paths to `BroadcastReceiver.abortBroadcast` from within an `onReceive` event handler. Confirmed by checking that `BroadcastReceiver.abortBroadcast` is invoked

③ **Deleting SMS**: Trigger paths to `ContentProvider.delete` where the URI is `content://sms`. Confirmed when a deletion occurs on the SMS content provider, where the deleted message was injected by IntelliDroid.

④ **Leaking information via SMS**: Trigger paths with calls to `sendTextMessage`. Confirmed by inspecting the content of messages sent by `sendTextMessage`.

⑤ **Network access**: Trigger paths to HTTP API methods. Confirmed by recording and inspecting the device's network traffic.

⑥ **Reflection and dynamic class loading**: Trigger paths to reflection and dynamic class loading API methods (e.g. `DexClassLoader.loadClass`). Confirmed by checking that the API methods are invoked.

In some cases, the malware constraints depend on values obtained from network requests to a remote control server. To resolve these constraints, IntelliDroid will monitor these network requests to extract the necessary values and solve the constraints to generate inputs that will match these requests. However, for the *CoinPirate*, *Crusewin*, and *Pjapps* malware, the third-party servers were no longer available and the network data values could not be extracted. To test these samples, we implemented an HTTP proxy server that imitates the original control server and responds to application requests with appropriately formatted replies.

Using the malicious dataset and the specification of targeted APIs, we measure the number of instances where IntelliDroid successfully generates inputs that trigger the targeted API. Many of the malware samples have multiple malicious invocations of the APIs, in which case they are tested once for each invocation. Table V provides detailed information

TABLE V. EFFECTIVENESS BY MALWARE FAMILY

Rows indicate the malware family and columns indicate the type of input(s) injected. Numbers indicate the type(s) of malicious activity triggered/missed.

Event →	SMS	Intent	Loc	UI	Life
AnserverBot	[2] [5] [6]			⑤ ⑥	⑥
Backflash	[1] ② ④ ⑤	⑤			⑤
Bgserv	② ④			①	① ⑤ ⑥
Cajino		① ⑤			⑤
CoinPirate	① ② ⑤				⑤
Crusewin	① ③ ⑤				⑤
DogWars		①			
Endofday	① ③ ④				
Fakemart	② ③				①
FakeNetflix				⑤	
FakePlayer					①
GamblerSMS	④				
GGTracker	① ② ⑤				⑤
GoldDream	[4]				⑤
GPSSMSSpy	④		④		
HeHe	① ② ⑤				⑤
Jifake					6 → 1
HippoSMS	① ②				
KMin	②			③	
NickyBot	③ ④				⑤
NickySpy		④			
Pjapps	② ④				⑤
RogueSPPush	① ② ③				⑤
SMSReplicator	① ④				
Spitmo	② ④				
Zitmo	⑤				
Zsone	① ②			①	

(*) Malicious behavior triggered [*] Missed behavior

about the targeted APIs found and triggered in each malware family. The specific targeted API (and their corresponding numbers) are described above and the table is organized with respect to the event handler that triggers the API. In the case of the *Jifake* malware, IntelliDroid extracted a path to a reflected method call and when dynamically executing this path, the reflected call triggered a malicious behavior that leaked sensitive information via SMS. Since the malicious behavior was triggered, a dynamic analysis tool would detect it in theory, even though IntelliDroid only generated inputs that triggered the path to the reflected call.

IntelliDroid was successful in triggering the targeted API in 70 out of the 75 instances. We found that IntelliDroid's ability to extract event-chains, perform device-framework input injection, and solve constraints at run-time were significant in achieving targeted execution, which we discuss below.

Event Chains: The event-chains generated by IntelliDroid were instrumental in 6 cases of malicious behavior and ensured that all constraints imposed by the application were satisfied. For example, in the *Endofday* and *Zsone* malware, the malicious behavior was activated only when the injected event occurs on a certain date or only after the application has been running for certain amount of time. In these cases, simply injecting a single event would not have satisfied the multi-event constraints that these applications impose. The event-chain for these paths includes a separate event to change the device's system time, which is triggered prior to the injection of the input that finally triggers the targeted API. In a different case, the *GPSSMSSpy* malware watches for a control SMS message that contains a certain string ("how are you?"). Once received, it saves the originating address of the message and begins listening for location updates. For each location update, it sends the location information to the saved SMS address, thereby leaking sensitive data to a third-party. IntelliDroid's event-chain generation component successfully recognizes this data-flow dependency through the third-party address saved on the heap and accessed in the location event handler, thus ensuring that the SMS is injected prior to the location event.

Device-Framework Injection: Injection at the device-framework interface was necessary in 9 cases of malicious activity. For these cases, the malware would not have behaved realistically if IntelliDroid had not implemented consistent framework behavior by injecting inputs into the framework rather than at the application boundary. For example, the *GamblerSMS* malware receives new SMS notifications by registering a custom `ContentObserver` object to listen for SMS database changes. Merely injecting new SMS events at the framework-application boundary would not have triggered the malicious behavior, since the injected event would not have been entered into the framework's SMS database. The *CoinPirate* and *HippoSMS* malware also use a similar technique to receive new SMS notifications while avoiding traditional telephony APIs, which are commonly detected. For these applications (and any others that use `ContentObserver` for other databases), it is essential that the events are injected at the device-framework interface so that they are entered into the appropriate database. In addition, 5 cases were found where SMS entries are deleted from the database when the application detects that a new SMS message was received. Often, these deletions require a query into the database to obtain a handle (e.g. URI) on the message. If the SMS event was not entered into in the database, the query would have failed and the deletion would not have been executed. If this occurred while screening the applications for malware, it would have caused the screening tool to miss potential malicious activity.

Run-time Constraint Data: Run-time data was required for the extracted constraints of 22 malicious call paths. This data included controlled variables such as the device time or location, and monitored variables derived from third-party server replies to network requests. For instance, the *CoinPirate*, *Crusewin*, and *Pjapps* malware contained malicious call paths relying on values obtained from a third-party server. For the malicious activity to occur, the network reply values are compared against those from the injected event and thus, the extracted constraints depend on the run-time variables. In other cases, values are obtained from the application's `SharedPreferences` file or from the device state. Because IntelliDroid employs a hybrid system and performs the constraint solving in the dynamic component, the statically extracted constraints could be augmented with the run-time data prior to generating the input values. Without the run-time variables, the constraints would not have been as precise and the malicious call path would not have executed fully.

Of the five malicious behaviors that IntelliDroid could not trigger, three of them occur for *AnserverBot*, which has path constraints that contain hash functions. The solving of

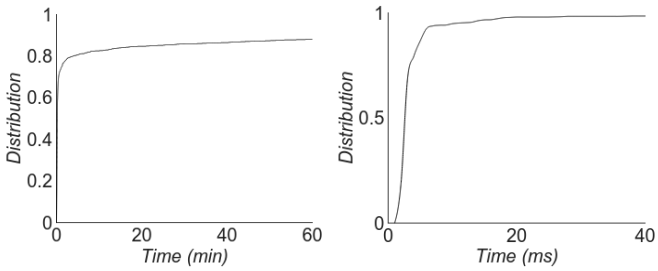
constraints containing hash functions is beyond the capabilities of the Z3 constraint solver that IntelliDroid uses. Similarly, *Backflash* contained constraints that require Base64 decoding and string/array manipulation, which IntelliDroid currently does not fully handle. The remaining case occurred in *GoldDream* and was the result of data flow through files. While IntelliDroid currently does not support flow through files, it would be possible to extend it to recognize file system dependencies in the same manner as heap dependencies.

C. Performance

We measured two quantitative performance aspects of IntelliDroid. The first is the reduction in analysis time IntelliDroid imparts by saving a dynamic analysis tool from having to exercise irrelevant portions of the application. For this, we measure the percentage of the application that IntelliDroid actually dynamically executes to allow TaintDroid to detect all privacy leaks. The second is the time IntelliDroid takes to generate and inject inputs, which has two distinct phases: (1) static extraction and analysis of path constraints; and (2) dynamic generation of inputs based on run-time state and constraint solving. We do not include the time to actually run the dynamic analysis as this is more of a function of the dynamic tool than of IntelliDroid.

Our previously described experiments with Intelli-TaintDroid give a glimpse of the reduction in analysis time that IntelliDroid can provide. While Monkey injected over 60K inputs, it was only able to trigger 7 of the 26 malicious behaviors that IntelliDroid could trigger with an average of 72 inputs. However, Monkey is a fairly simplistic tool and it was not possible to integrate more complex tools with TaintDroid. Thus, we measure the average percentage of application code that Intelli-TaintDroid must exercise and compare against the amount of code that an input generator based on random fuzzing or concolic testing might need to execute to achieve the same detection results. By measuring the total number of call graph nodes and edges in each application and comparing with the number that IntelliDroid actually executes, we find that IntelliDroid need only execute less than 5% of the code on average in the applications we tested. On the other hand, both random fuzzing and concolic testing, which inject inputs without being aware of the goals of the dynamic analysis, might have to statistically execute 50% or more of the application before it has a better than 50% probability of trigger all the relevant behavior in an application. This conservatively suggests that IntelliDroid might cut execution time by as much as 90% against state-of-the-art input generation methods, and this estimate does not take into account that the number of paths (and thus inputs) is actually exponentially related to the size of the code. In addition, fuzzing and concolic testing do not actively determine the correct order in which inputs must be injected so they may have to try several permutations to achieve full coverage.

IntelliDroid's static analysis time and the number of inputs it must inject is heavily dependent on the number of target APIs specified. Thus, to simulate a worst-case scenario with a very comprehensive dynamic analysis engine, we use an even larger set of targeted APIs than in our previous experiments by deriving them from the set of potentially malicious APIs identified by the FlowDroid static analysis tool [4]. The fact



(a) Static Analysis (b) Constraint Solving

Fig. 2. IntelliDroid Timing Distribution

that FlowDroid uses static analysis is not relevant – we use this set mainly because it is a large collection of Android APIs that have been identified as potentially malicious. We note that this set of targeted APIs is a superset of both the targeted APIs used by TaintDroid and the targeted APIs used by the hypothetical malware detection tool in our experiments, containing a total of 228 API methods. The extra methods in the FlowDroid set include more conservative sources and sinks, such as those where data is sent via an intent or printed in a log message. These generic API methods are commonly used by both malicious and benign applications.

We measure the time IntelliDroid takes to find invocations of the targeted APIs and extract the constraints required for input generation using a combination of two datasets: 1260 malware samples from the Android Malware Genome project [49] and 1066 benign applications from the Android Observatory [8]. The Android Observatory dataset was obtained by filtering for applications that declare the permissions necessary for the set of targeted APIs used in this experiment. The static component, running with a time limit of 60 minutes (enforced for timeliness of results), took an average of 138.4 seconds per application and 88.1% completed analysis within the time limit, with the distribution shown in Figure 2a. The bigger set of targeted APIs and the larger applications in the benign dataset used in this experiment resulted in an average of 1760 inputs generated for each benign application. Despite this large number, the extracted paths still comprise less than 5% of the code in each application on average.

The analysis time of IntelliDroid is dominated by WALA's call graph extraction and the search for targeted API invocations, which must be performed on the entire application and accounts for roughly 50% of the static analysis time. We found that the applications that required longer analysis times often used advertisement libraries. The extra code included with these libraries resulted in larger call graphs and thus, more time was spent searching for targeted APIs.

Unlike the static component, the dynamic generation of input values must be extremely quick since it is performed for every injected input, of which there could be several thousand per application. Because the constraint solver component of IntelliDroid is completed during run-time, it is especially important that it runs efficiently. We measure the total time taken by the Z3 solve the constraints for all target paths in our dataset to be an average of 4.22 milliseconds, with the distribution show in Figure 2b. As a result, we expect the main run-time cost to be that of the dynamic analysis tool itself.

VI. Limitations

A. Call-Graph Generation

Since IntelliDroid aims to generate inputs to event handlers that will trigger all targeted APIs, the tool needs an accurate call graph to be extracted. Missing edges in the call graph may cause IntelliDroid to incorrectly believe that a targeted API is not reachable and thus cannot be triggered by any inputs. This is particularly challenging for Android because there are many implicit paths that applications can take via intents, callbacks and other Android-specific facilities. This challenge is not limited to IntelliDroid, but is common to all tools that perform any sort of static analysis on Android applications. We currently model all Android-specific call edges that we are aware of with the exception of exceptions, although we can report exception handlers that invoke a targeted API as a case we cannot handle and flag it as suspicious. We did not encounter any such cases in our experiments.

Another limitation of our prototype, documented in Section V, is that IntelliDroid currently only detects inter-event data dependencies if they occur through heap variables. Thus, IntelliDroid failed to generate inputs for data flow through a file. It would be straightforward to extend dependency tracking through files, as well as other Android facilities such as content providers, in the same manner.

B. Generating Constraints

Most technical limitations are the result of limitations of the constraint solver used. In some cases, the extracted constraints contain theories that are undecidable with current solvers, such as trigonometric functions to compute location changes. In other cases, the extracted constraints are too complex for the constraint solver, such as the functions that convert the SMS PDU bytecode format [1] used by the hardware for the SMS message format. IntelliDroid mitigates such shortcomings by extracting the necessary information at run-time and solving for inputs dynamically, but this currently still requires manual instrumentation of the Android framework. Fortunately, this instrumentation need only be done once for each Android version (or even several versions, since system service interfaces rarely change) and once done, the results can be re-used without modification for all applications that are analyzed.

Another inherent limitation of inputs generated through constraint solving is that they are not necessarily realistic and thus might not happen in practice. For example, IntelliDroid can manipulate time and other inputs in such a way that it injects a sequence of inputs that is not physically possible, resulting in the detection of malicious behavior that cannot happen in reality. We see no reason why IntelliDroid cannot be enhanced to account for the constraints of the physical world when generating inputs. The main challenge would be enumerating what all the relevant physical constraints are.

C. Malware Obfuscation

More recent malware may try to obfuscate their behavior by using reflection and encryption. While IntelliDroid has been implemented with support for constant reflection targets during call graph generation, in general, IntelliDroid can only compute inputs that will trigger the reflection. It cannot determine path constraints after the reflected call because the statically extracted call graph is incomplete at that point. A possible solution that we are currently exploring is to feed dynamic information back to the static component to resolve such issues and build a more complete call graph. In particular, IntelliDroid's targeted execution can be used to ensure that these statically unresolved method invocations are executed during run-time to obtain the dynamically resolved target.

A similar and related limitation is that IntelliDroid is unable to compute inputs that are processed by complex functions (e.g. encryption or hashing) in a path constraint. This is because constraint solvers are generally unable to determine the inputs to such functions, which are necessary to produce an output that would satisfy the path constraint. Again, in some cases, a system that uses dynamic feedback as described in the case of reflection might allow IntelliDroid to produce inputs in these situations, and we are currently exploring this approach.

Similar to obfuscation through reflection, malware developers may hide malicious behavior in packed applications or native code, which IntelliDroid does not support. IntelliDroid can be used to direct execution to locations in the code where these are used (e.g. `DexClassLoader`, JNI invocations); however, once again, any constraints that occur after these invocations will not be extracted by the static component. While not ideal, the ability to direct execution to these questionable parts of the application is still valuable and can help an attached dynamic tool analyze these portions more effectively.

D. Knowledgeable Attacker

Given the above limitations, a suitably knowledgeable attacker has two main avenues for defeating IntelliDroid. First, she can exploit the technical difficulty of extracting a complete call graph for Android applications by placing the malicious code in a section of code that appears to be disconnected from the rest of the call graph (i.e., dead code). Since IntelliDroid cannot determine a path to the code, it cannot generate inputs. The mitigations for this is to have more precise modeling of Android call edges, as well as conservative over-approximation of call edges. The former requires more engineering effort, while the latter may result in IntelliDroid injecting inputs for paths that are not actually possible to execute.

Second, the attacker can process inputs in malicious code with complex functions such as encryption and cryptographic hashing that will defeat the current generation of constraint solvers (and likely many future ones as well). In such cases, however, IntelliDroid will experience many constraint solver time-outs, which in itself is anomalous as none arose during our experiments. While not necessarily indicative of malicious behavior, they are infrequent enough to certainly warrant more attention and possibly manual analysis.

VII. Related Work

Static analysis of Android applications has been widely used to detect malicious behavior or vulnerabilities [22], [25], [23], [29], [4], [16], [30]. IntelliDroid's static analysis is comparable to the techniques used in previous work, though in some cases it reduces precision for better scalability and analysis speed.

IntelliDroid is designed to complement dynamic analysis tools to allow them to quickly identify and analyze paths that are likely to contain malicious behavior. There are a variety of dynamic analysis tools that IntelliDroid could be used with, such as TaintDroid [19], CopperDroid [39], DroidScope [44], VetDroid [47] or RiskRanker [27]. Similarly, IntelliDroid can also be used to aid reverse-engineering or manual analysis using sandboxing analysis tools such as DroidBox [18].

While IntelliDroid's extraction of path constraints is technically a form of symbolic execution, it is performed on a static abstraction of the program rather than on a concrete execution trace. As a result, it should generally provide faster performance than concolic test generation systems such as Dart [26], EXE [15] and KLEE [14], which use concrete symbolic execution. In addition, IntelliDroid's main focus is on generating inputs to trigger a specific path rather than obtaining code coverage, making its goals fundamentally different from these systems, as well as more recent Android-focused concolic testing work, such as DynoDroid [31] and the ACTEve algorithm [2]. The work in [32] targets malicious code by exploring paths that branch on interesting input, although the input dependency tracking and constraint extraction is performed dynamically. Purely static constraint extraction and solving has been used in tools like Saturn for verification [43] and hybrid static/dynamic symbolic execution is used in MergePoint [7]. IntelliDroid is also similar to AEG [6], APEG [11], and DyTa [24] which generate malicious inputs to exercise vulnerabilities in program binaries. However, these systems do not target Android applications and thus, do not have to handle consistent input injection or event-chains.

The work most closely related to IntelliDroid are hybrid static/dynamic analyses such as AppAudit [42], ContentScope [28], AppIntent [45], SmartDroid [48], Smv-Hunter [37] and Brahmastra [9]. The main difference between IntelliDroid and these systems is the level of fidelity of the injected inputs. IntelliDroid can inject inputs into an actual Android system, enabling integration with full system dynamic analysis tools such as TaintDroid [19]. To do this, it must detect event-chains and perform device-framework input injection. In contrast, systems like AppAudit and ContentScope rely mainly on the static analysis to find vulnerabilities, and only use dynamic analysis to confirm the feasibility of the paths. Moreover, ContentScope focuses solely on content providers. In contrast, IntelliDroid's goal is to detect malware so it must support and analyze a wider range of behavior. AppIntent also uses static analysis to identify relevant sections of code to execute. However, while IntelliDroid targets specific paths and statically generates concrete inputs, AppIntent requires an exhaustive dynamic symbolic execution to fully explore all behaviors, similar to that used in concolic testing. In addition, AppIntent, SmartDroid, Brahmastra, and Smv-Hunter only handle UI events.

VIII. Conclusion

IntelliDroid is a targeted input generator that specifically exercise code paths in an application that are relevant to a dynamic analysis tool. This paper contributes several novel ideas that enable IntelliDroid to achieve this goal, such as the use of targeted APIs as an abstraction for dynamic analysis techniques, event-chain detection and input generation, and device-framework injection. Using our prototype, we find that the static analysis component can identify and generate inputs to trigger the targeted behavior in less than 138.4 seconds on average. The generated inputs are able to trigger 70 out of 75 malicious behaviors in a set of malware, while saving the dynamic analysis from having to execute 95% of the application code. When integrated with TaintDroid [19] and compared against FlowDroid [4], we find that IntelliDroid-targeted TaintDroid is able to offer better precision. A comparison with Monkey shows that IntelliDroid's targeted execution triggers malicious paths more precisely than a standard off-the-shelf input fuzzer.

Acknowledgment

We would like to thank Zhen Huang, Mariana D'Angelo, Dhaval Miyani, Wei Huang, Beom Heyn Kim, Sukwon Oh, and Afshar Ganjali for their suggestions and feedback. We also thank the anonymous reviewers for their constructive comments. The research in this paper was supported by an NSERC CGS-M scholarship, a Bell Graduate scholarship, an NSERC Discovery grant, an ORF-RE grant, and a Tier 2 Canada Research Chair.

References

[1] 3GPP, "Technical realization of Short Message Service (SMS)," 3rd Generation Partnership Project (3GPP), TS 23.040, Sep. 2014.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[3] "Apkparser," http://code.google.com/p/xml-apk-parser/, accessed: September 2014.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le 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*. ACM, 2014, p. 29.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2012.

[6] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation." in *Proceedings of the 18th Symposium on Network and Distributed System Security (NDSS)*, 2011, pp. 59–66.

[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1083–1094.

[8] D. Barrera, J. Clark, D. McCarney, and P. C. Van Oorschot, "Understanding and improving app installation security mechanisms through empirical analysis of android," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 81–92.

[9] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 2014, pp. 1021–1036.

[10] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Malicious and unwanted software (MALWARE), 2010 5th international conference on*. IEEE, 2010, pp. 55–62.

[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 143–157.

[12] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[13] J. Caballero, P. Poosankam, S. McCamant, D. Song *et al.*, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 413–425.

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

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[16] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[17] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] A. Desnos and P. Lantz, "DroidBox: An android application sandbox for dynamic analysis," 2014, https://code.google.com/p/droidbox/, Last Accessed Oct, 2014.

[19] W. Enck, P. Gilbert, 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," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010, pp. 1–6.

[20] Ericsson Mobility, "Interim update: Ericsson mobility report," Feb. 2015, http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-feb-2015-interim.pdf.

[21] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[22] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011, pp. 627–638.

[23] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/~ avik/projects/scandroidascaa*, 2009.

[24] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "Dyta: dynamic symbolic execution guided with static verification results," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 992–994.

[25] C. Gibler, J. Crussell, J. Erickson, and H. Chen, *AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale*. Springer, 2012.

[26] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[28] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[29] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, 2012.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[31] A. MacHiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[32] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 231–245.

[33] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to java bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 6.

[34] M. Parkour, "Contagio mobile," 2015, http://contagiominidump.blogspot.ca/, Last Accessed Aug, 2015.

[35] "Robotium," 2014, https://code.google.com/p/robotium/.

[36] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[37] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proceedings of the 19th Network and Distributed System Security Symposium*, 2014.

[38] V. Svajcer, "Sophos mobile security threat report," 2014, http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf.

[39] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[40] "Testing fundamentals," 2014, http://developer.android.com/tools/testing/testing_android.html, Last Accessed Oct, 2014.

[41] "Watson libraries for analysis," http://wala.sourceforge.net, accessed: September 2014.

[42] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. IEEE Computer Society, 2015.

[43] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, p. 16, 2007.

[44] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *Proceedings of the 21th USENIX Security Symposium*, 2012, pp. 569–584.

[45] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 1043–1054.

[46] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 2013, p. 68.

[47] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 611–622.

[48] 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 *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[49] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[50] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.