SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution

Chaoshun Zuo
The University of Texas at Dallas
800 W. Campbell RD
Richardson, Texas
chaoshun.zuo@utdallas.edu

Zhiqiang Lin
The University of Texas at Dallas
800 W. Campbell RD
Richardson, Texas
zhiqiang.lin@utdallas.edu

ABSTRACT

Server URLs including domain names, resource path, and query parameters are important to many security applications such as hidden service identification, malicious website detection, and server vulnerability fuzzing. Unlike traditional desktop web apps in which server URLs are often directly visible, the server URLs of mobile apps are often hidden, only being exposed when the corresponding app code gets executed. Therefore, it is important to automatically analyze the mobile app code to expose the server URLs and enable the security applications with them. We have thus developed SMARTGEN to feature selective symbolic execution for the purpose of automatically generate server request messages to expose the server URLs by extracting and solving user input constraints in mobile apps. Our evaluation with $5,000$ top-ranked mobile apps (each with over one million installs) in Google Play shows that with SMARTGEN we are able to reveal $297,780$ URLs in total for these apps. We have then submitted all of these exposed URLs to a harmful URL detection service provided by VirusTotal, which further identified $8,634$ URLs being harmful. Among them, $2,071$ belong to phishing sites, $3,722$ malware sites and $3,228$ malicious sites (there are 387 overlapped sites between malware and malicious sites).

Keywords

Mobile App, Symbolic Execution, URL Security

1. INTRODUCTION

Over the past several years, we have witnessed a huge increase of the number of mobile devices and mobile apps. As of today, there are billions of mobile users, millions of mobile apps, and hundreds of billions of cumulative mobile app downloads [6]. When talking to a remote service (e.g., user registration, login, password reset, and pushing and pulling data of interest), mobile apps often use URLs which include domain (or host) names, resource path, and request parameters. Unlike traditional desktop web apps in which the server URLs are directly visible to the end users and web browsers (e.g., they are in the address bar), the server URLs of mobile apps are often hidden in the mobile apps, and only when the corresponding app code is executed does it become visible.

By making URLs invisible to app users has certainly made the mobile app more user friendly. For instance, a user does not have to remember the server address and enter the URLs to access the mobile services. Unfortunately, it has also introduced a number of security issues. First, hiding the URLs may allow the servers to collect some private sensitive information (e.g., GPS coordinates and phone address books can be sent to the servers at certain URLs). Second, it may also allow the mobile apps to talk to some unwanted services (e.g., malicious ads sites). Third, it can also provide the illusions to the app developers that their services are secure (security through obscurity), since their server URLs are hidden, none knows and none will attack (or fuzz) them.

Therefore, exposing the server URLs of mobile apps are important to many security applications such as hidden service identification, malicious website detection, and server vulnerability hunting. However, we must analyze the app code to expose them since they are often fragmented and scattered (e.g., a domain name can appear here, but the request parameters or the resource path can be there). Interestingly, as the URLs must be used in the network communication APIs, we can perform a targeted analysis: namely starting from these APIs, we can infer and observe how the parameters are generated and used, to reveal the URLs.

Recently, there are already a group of efforts in the direction of targeted code analysis of mobile apps. Specifically, A3E [13] performs taint-style data flow analysis to build a high level control flow, from which it performs a targeted exploration of app activities, but it does not attempt to solve the path constraints, which can stop the direct exploration of certain path. AppsPlayground [25] and SMV-Hunter [28] recognizes the labels in the UI elements, using them to more intelligently generate user input, but it still does not provide any soundness or completeness guarantees. While symbolic execution has also been explored, existing efforts either focused on capturing and solving the constraints for activity transitions (e.g., ACTEve [10] which unfortunately needs to access app's source code), or only targeted for malicious app analysis (e.g., IntelliDroid [29]) which has little UI involvement.

To advance the state-of-the-art, we develop SMARTGEN, a new targeted, symbolic execution enabled tool to automatically explore the UI of a mobile app with the goal of systematically exposing the server URLs. Similar to many of the existing approaches (e.g., [16, 29]), we also build an extended call graph (ECG) based on the APK code (not the source code) of an app for each entry point of app execution; such a graph captures not only the explicit function calls but also the implicit ones introduced due to the Android framework callbacks. Guided by the ECG, we then traverse the graph to locate whether there is any invocation of network message sending

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052609

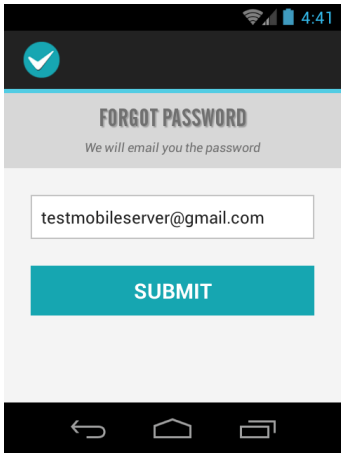

Figure 1: The password reset activity of ShopClues.

APIs. If so, we extract the corresponding path constraints that control the execution of these APIs. We then solve these constraints and execute the app with a new dynamic runtime instrumentation technique we developed, to control the app executed in a real smartphone, provide proper input, and explore the possible network message sending activities. Eventually, the execution of each network message sending API will generate a server request message which usually contains the server URLs.

With the revealed URLs and generated server request messages (which show how many parameters are involved and what kind of server interface would process the user request), we can feed them as seeds to the existing automatic server vulnerability hunting tools (e.g., `sqlmap` [1] for SQL injection, or `watcher` [3] for cross-site-scripting) to fuzz the server, if we have the permission to do so. We can also use them to detect whether there is any hidden services or malicious URLs. In this paper, we focus on harmful URL detection, and we have tested SMARTGEN with $5,000$ top ranked Android apps (each with more than one million installs) crawled from the Google Play. Our evaluation shows that with SMARTGEN we were able to reveal $297,780$ URLs in total for these apps, whereas using non symbolic execution tool such as Monkey can only reveal $128,956$ URLs. By submitting all of these exposed URLs to a harmful URL detection service at VirusTotal for security analysis, we have obtained $8,634$ harmful URLs. Among the $297,780$ reported URLs, 83% of them are the first time submitted to VirusTotal.

In summary, we make the following contributions:

- We propose selective symbolic execution, a technique to solve input-related constraints for targeted mobile app execution. We also develop an efficient runtime instrumentation technique to dynamically insert analysis code into an executed app in real mobile devices using API hooking and Java reflection.
- We have implemented all of the involved techniques and built a novel system, SMARTGEN, to automatically generate server request messages and expose server URLs.
- We have tested SMARTGEN with $5,000$ top-ranked Android apps and exposed $297,780$ URLs in total. We found these top ranked apps will actually talk to $8,634$ malicious and unwanted web services, according to the harmful detection result from VirusTotal.

```
PUT /api/v9/forgotpassword?key=d12121c70dda5edfgd1df6633fdb36c0
HTTP/1.1
Content-Type: application/json
Connection: close
User-Agent: Dalvik/1.6.0 (Linux; Android 4.2)
Host: sm.shopclues.com
Accept-Encoding: gzip
Content-Length: 73

{"user_email":"testmobileserver@gmail.com","key":"d12121c70dda5e
dfgd1df6633fdb36c0"}
```

Figure 2: The password reset message sent by ShopClues.

2. BACKGROUND AND RELATED WORK

2.1 Objectives

The goal of this work is to expose the server interface (namely the URLs) of mobile apps by generating server request messages from the app. While there are many ways to do so, we wish to have an approach that is:

- **Automated**. The system should not involve any manual intervention, such as manual installation and manual launching, and instead everything should be automated.
- **Scalable**. Since we aim to expose the URLs for apps in popular app stores such as Google Play, we need an approach that can perform a large scale study. For instance, it should not take too much time to generate a server request message.
- **Systematic**. The path exploration should be systematic. We should not blindly click a button or randomly generate an input to explore the app activities. Instead, all the targeted paths containing the network message sending APIs should be explored.

2.2 A Running Example

To illustrate the problem clearly, we use a popular app ShopClues as a running example. This app has between 10 million and 50 million installs according to the statistics in Google Play. There are many activities inside this app, and for simplicity we just use its password reset activity, as shown in Figure 1, to describe how we would have performed our analysis.

In particular, if we aim to reveal whether the password reset interface at the server side of ShopClues contains any security vulnerabilities (e.g., SQL injection), we need to enter a valid email address in the corresponding `EditText` box, and then click the `SUBMIT` button, which will automatically generate a sample password reset request message as shown in Figure 2. With this message, the server interface (e.g., the host name, the query parameter) of password reset is clearly exposed. Next, if we have the permissions from the service provider to perform penetration testing, we may apply the standard server vulnerability fuzzing tools such as `sqlmap` [1] to automatically mutate this request message (e.g., adding some SQL commands such as "`and 1=1`" to the request fields and analyze the response message to check whether the SQL commands get executed) to determine whether the server contains any testable security vulnerabilities.

However, it is non-trivial to generate a sample request message to expose the URLs as shown in Figure 2. Specifically, it requires a valid input in the `EditText` box, and meanwhile a click of the `SUBMIT` button. In fact, there are also a number of checks of the `EditText` input, as shown in the sample decompiled code in Figure 3 for the `SUBMIT onClick` event. More specifically, to really trigger the password reset request, the user input of `EditText` has to pass the check of none empty (line 8) and match with an email address format (line 33), while the app needs to maintain a

```
1  package com.shopclues;
2
3  class y implements View$OnClickListener {
4      EditText b;
5      ...
6      public void onClick(View arg5) {
7          String v0 = this.b.getText().toString().trim();
8          if(v0.equalsIgnoreCase("")) {
9              Toast.makeText(this.a, "Email Id should not be
               empty", 1).show();
10         }
11         else if(!al.a(v0)) {
12             Toast.makeText(this.a, "The email entered is not
               a valid email", 1).show();
13         }
14         else if(al.b(this.a)) {
15             this.a.c = new ac(this.a, v0);
16             this.a.c.execute(new Void[0]);
17         }
18         else {
19             Toast.makeText(this.a, "Please check your
               internet connection", 1).show();
20         }
21     }
22 }
```

```
23 package com.shopclues.utils;
24
25 public class al {
26     ...
27     public static boolean a(String arg1) {
28         boolean v0;
29         if(arg1 == null) {
30             return false;
31         }
32         else {
33             v0 = Patterns.EMAIL_ADDRESS.
                   matcher(((CharSequence)arg1)).matches();
34         }
35         return v0;
36     }
37 }
```

Figure 3: The decompiled code of the `onClick` event handler for the password reset request in ShopClues.

network connection (line 16). Without a correct email address and network connection, it is impossible to expose the password reset server URL.

While the path constraint appears to be a bit simple in our running example, there are other sophisticated ones, such as when the contents of two `EditTexts` need to be equivalent (e.g., when registering an account, the confirmed email address needs to be equivalent to the one entered first), when the age needs to be greater than 18, when a zip code needs to be a five digit sequence (a phone number may also have similar checks), when a file name extension needs to match some particular pattern (e.g., `jpg`), etc. We thus need a systematic approach to solve these constraints and expose the server request messages including the URLs.

2.3 Related Work

Static Analysis. Static analysis is often scalable since it does not have to execute the app. However, we have to exclude those purely static analysis systems. This is because what we need is a concrete request message (which are often the seeds for a fuzzing tool such as `sqlmap`). One sample of such a message is illustrated in Figure 2, and we find that it is very challenging to statically generate such a request message, because we have to solve the following challenges:

- **String Concatenation**. We need to concatenate some strings. For instance, a URL often contains a domain name (e.g., `sm.shopclues.com`) and a path (e.g., `/api/v9/forgotpassword`) and some user input as parameters to the service. Different request messages often need to pass to different server paths with different parameters. Only when certain behavior gets triggered, can we concatenate the corresponding path with the corresponding parameters; but it is quite challenging to achieve this via static analysis.
- **Field Recognition and Value Generation**. When sending a request message to a server, there are often several fields such as `user_email` and `key` as shown in our running example. Their values are also context specific. e.g., `key` needs to be `d12121c70dda5edfgd1df6633fdb36c0`. How to identify these values statically is also a challenge.
- **Data Format Recognition**. The app also needs to package the user input in a request message using certain format, such as using `json` as in our running example or other formats such as `xml`. We must also determine them when statically generating the request messages.

Dynamic Analysis. We can avoid solving all the static analysis challenges if we can execute the app directly and use dynamic analysis. Recently, there are a set of dynamic app testing tools such as Monkey [7] that can automatically execute and navigate an app when given only a mobile app binary, or Robotium [4], a testing framework for Android apps that is able to interact with UI elements of an app such as menus and text boxes.

There are also more advanced systems beyond just simply interacting with the UI elements. AppsPlayground [25] and SMV-Hunter [28] recognize the UI elements and generate text inputs in a more intelligent way. A3E [13] performs a targeted exploration of mobile apps guided by a taint-like static data flow analysis. DynoDroid [21] instruments the Android framework and uses the debugging interface (i.e., the `adb`) to monitor the UI interaction, and guide the generation of UI events for app testing. PUMA [18] provides a programmable interface for large scale dynamic app analysis. DECAF [20] and VanarSena [27] navigate various activities of Windows phone apps and seek to detect ads, flaws, or debug crashes. Brahmastra [14] efficiently executes third party components of an app to expose its potential harmful behavior such as private data collection via binary rewriting.

However, we still cannot directly use them for our purpose. Specifically, each system is designed for a particular application scenario with different goals: A3E for bug detection, SMV-Hunter for SSL/TLS man-in-the-middle vulnerability identification, DECAF for ads flaw issue (recent work also applied AppsPlayground for this detection [26]), VanarSena for crash debugging, and Brahmastra for targeted vulnerability (e.g., privacy leakage or access token hijacking) identification in 3rd party components of the local app. None of them focused on the remote server request message generation.

More importantly, aside from A3E, which runs in real devices, all of these systems run in an emulator, which often has several limitations. For instance, emulator lacks physical sensors, and it cannot provide a high fidelity environment for testing the app. Second, emulation is slow and it often takes a long time to boot and restart, and sometimes is also unstable. Therefore, we would like to directly execute the testing apps in real phones. While A3E uses the real phone, the exploration of the app activity can fail since it does not attempt to solve any constraints for more targeted execution.

Symbolic Execution. Being a systematic path exploration technique, symbolic execution has been widely used in many security applications (e.g., vulnerability identification [15], malware analysis [24], and exploit generation [12]). Recently, there have also been efforts to apply symbolic execution to the analysis of mobile apps for various applications, such as app testing in general [23],

path exploration [10], and malware analysis [29]. However, some of them require access to app source code, which is impractical in our application, and only IntelliDroid [29] directly works on app binary (Java byte code, essentially) during the symbolic execution.

While we may directly use IntelliDroid [29] to solve our problem, it is not suitable after examining its detailed design as well as the corresponding source code. Specifically, IntelliDroid does not attempt to perform any UI analysis since malware tends to have significantly fewer UI elements in its interface compared to normal apps. Consequently, IntelliDroid does not have to capture the constraints from UI elements. Second, IntelliDroid does not precisely inject an event (e.g., a particular button click). Instead, the events injected by IntelliDroid are at the system boundary, and an injected event may not be accurately delivered to the target app. Third, IntelliDroid requires the instrumentation of the Android framework, but such instrumentation needs to run in an emulator, meaning certain app behavior may not get exposed. Therefore, we have to design our own symbolic execution and leverage the existing efforts, including those dynamic analyses, to automatically generate server request messages.

3. OVERVIEW

Scope and Assumptions. We focus on Android platform, and analyze the apps that use HTTP/HTTPS protocols. Note that according to our evaluation, all of the tested mobile apps use at least HTTP/HTTPS protocol once (there may be more than one protocols in a given app). We assume the app is not obfuscated and can be analyzed by Soot [2], a general Android APK analysis framework. Also, we primarily focus on string constraints since they are often user input related. Other non-linear, non-solvable constraints by standard solvers are out of our scope.

Challenges. While the use of dynamic analysis and symbolic execution has avoided solving many practical challenges such as recognizing the protocol fields and automatically generating the request messages, we still encounter a number of other challenges:

- **How to instrument the analysis code**. When given a mobile app, we need to analyze its UI for each activity, provide a proper input (such as a valid email address in our running example), and inject a corresponding event (such as the SUBMIT button click) to the app, to trigger the server request messages. These analysis behaviors are often context sensitive, and only after the activities are created can we trigger them. Therefore, we have to instrument the original app with context-sensitive analysis code.
- **How to extract the path constraints**. Not all the app execution paths are related to our server request message generation, and instead we are only interested in the path constraints that are related to the final message sending events. Therefore, we have to identify the invocation of network message sending APIs, their path constraints that are controlled by the input, and then solve them to generate the proper input. Also, the input to SMARTGEN is just the APK, we have to analyze the APK file to extract the constraints.
- **How to explore the app activities**. An app can have many activities (i.e., many single screens atop which containing various UI elements). The execution of one activity often determines the execution of others. At a given activity, we need a strategy to explore others (e.g., a depth-first search or breadth-first search). This also implies we need to know the follow-up activities at a given activity. However, this is also non-trivial since it requires sophisticated code analysis to resolve the target activities.

Solutions. To address the above challenges, we have developed the following corresponding solutions:

- **Dynamically instrumenting the apps in real phones**. While rewriting the Android system code including both the Java and native code is a viable approach to inserting the analysis code into a target app, this approach will introduce a system-wide change to all the apps, which may cause unstable behavior. Therefore, we propose a new approach to dynamically instrument the analysis code into the targeted app. At a high level, our approach uses API hooking to intercept the app execution flow dynamically, and then perform an in-context analysis and use Java reflection to manipulate the UI elements.
- **Extracting the path constraints of interest**. The execution of mobile apps are event driven, and we have to focus on the code that is of our interest. To this end, we first build an extended call graph (ECG) for the app that connects not only the explicit edges but also those implicit ones such as the call-backs. Starting from the network message sending APIs, we backward traverse the ECG, collect the path constraints, and meanwhile correlate the constraints with the user input if there is any. Then, we invoke a standard solver to solve the constraints. When the activity involving network message sending event is created, we initialize the UI elements with the proper input provided by the solver.
- **Exploring the app activities using DFS**. We need to explore the app activities as many as possible and in a systematic way. A given activity can trigger several other activities (e.g., based on different buttons a user has clicked). While we can use a breadth-first search (BFS) to explore all possible activities, we prefer a depth-first search (DFS) since our analysis has already reached a given activity and we can keep exploring a next layer activity further and then come back to explore the rest same layer activities recursively. Meanwhile, by hooking the `onCreate` event of a given activity, we can analyze all of its UI elements to determine whether there is any other next-layer activities associated with the current one and use this knowledge to guide our DFS activity exploration.

Overview. An overview of SMARTGEN is presented in Figure 4. There are three phases of analysis and five key components inside SMARTGEN.

- **Static Analysis**. The first phase of the analysis is to build the ECG, which contains all the possible function call transfer edges inferred statically. This is achieved by our *Building ECG* component, which takes the APK as input and produces the ECG as output.
- **Selective Symbolic Execution**. In our selective symbolic execution phase, the second and third components of SMARTGEN will extract the path constraints of interest based on the ECG and then solve the constraints if there is any by a constraint solver. The output in this phase will be the proper input value for each involved UI elements in each possible activity.

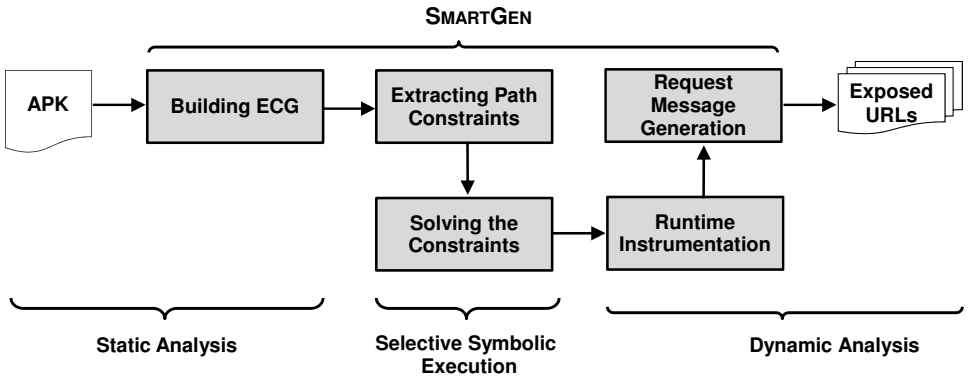

Figure 4: An overview of SMARTGEN.

- **Dynamic Analysis**. Finally, in the dynamic analysis phase, the last two components of SMARTGEN will instrument the app execution at runtime via API hooking, then initialize the UI elements with the value solved by the constraint solver, and finally generate the request messages. The output of this phase is the possible desired server request messages with exposed URLs.

4. DETAILED DESIGN

4.1 Building ECG

The goal of SMARTGEN is to generate the server request messages to expose the server URLs for a given app. Therefore, we should focus on the code path that finally invokes the targeted network message sending APIs. As such, we need to first build an extended call graph (ECG) of the Android app, which covers not only the explicit function call edges but also the implicit edges introduced by the call-back functions in Android framework, and then identify the code of our interest based on the ECG.

Since the input to SMARTGEN is just the APK, we should convert the APK into some intermediate representation (IR) suitable for our analysis. To this end, we use Soot [2], an Android app analysis framework that takes the APK as input and is able to perform various static analyses including call-graph construction, points-to analysis, def-use chains, and even taint analysis [17] in combination with FlowDroid [11]. It is thus a perfect framework for our ECG construction.

However, the call graph constructed directly from Soot IR will miss the edges that implicitly call the Android framework APIs. For instance, a thread `start` and threat `run` does not have an explicit call relation, but the Android framework uses a callback mechanism to ensure that the execution first starts from the `thread.start` and then `thread.run`. Therefore, we need to add these implicit calls. How to systematically identify all of them in the Android framework is another challenge. Fortunately, EdgeMiner [16] has been designed to exactly solve this problem. Specifically, EdgeMiner is able to analyze the entire Android framework to automatically generate API summaries that describe the implicit control flow transitions such as the callback relation of `thread.start` and `thread.run` pair in the Android framework, and we hence directly use the summary result to connect the implicit edges.

To build the ECG, we scan the primary IR (namely the Jimple IR) produced by Soot, and take each event handler (e.g., `onCreate`, `onResume`, `onClick`, `onTextChanged`, etc.) as a starting point, recursively add the callee edges if there is any, including those implicit ones guided by the summary produced by EdgeMiner. The output will be a set of ECGs, each starting from the event handler functions, since there are multiple such functions.

4.2 Extracting the Path Constraints

After we have built the ECG, then we traverse each ECG to determine whether there is an invocation of a network message sending API. If so, such an ECG is of our interest, and we then build a control flow path (according to the Soot IR) from the entry point of the ECG to the invocation point, from which to extract the path constraints.

The Targeted APIs. We focus on two sets of network message sending APIs. One is those provided by Android framework (e.g., `HttpClient.execute`), and the other is those low level `Socket` APIs (e.g., `Socket.getOutputStream`). With these functions as target, we traverse each ECG and identify the call path that invokes them. When these APIs get called, they will directly perform Internet connections to remote servers, which will then generate the desired request messages with the exposed URLs.

Taint Analysis. However, the path constraints of our focus are often user input related, and we have to correlate them with the input entered via the UI elements. To this end, we have to taint the inputs from the UI elements and track their propagations to resolve their proper values. There are already public available tools such as FlowDroid [11] for the taint analysis and also the Soot framework supports the integration with FlowDroid, and we thus design our taint analysis atop FlowDroid using Soot. Since taint analysis is a well-established area, below we just briefly describe how we customize FlowDroid's taint analysis for our purpose.

- **Taint Sources**. The taint sources in our analysis are those user input related UI elements such as `EditText` (an editable text field as showning in our running example), `CheckBox`, `RadioButton`, `ToggleButton`, `Spinner` (a drop-down list), etc. We thus assign a unique index tag for each of these UI elements as the taint tag and propagate the tag when necessary.
- **Taint Propagation**. Since we use the FlowDroid taint analysis framework, we do not have to customize the taint propagation rules. At a high level, a taint tag will be propagated based on the direct data flow propagations according to the Soot IR (therefore those implicit taint propagation will also be out of our focus).
- **Taint Sinks**. The taint sinks are the data use point of the tainted variables at the `if-stmt` in Soot IR (note that a

loop statement is implemented using `if` and `goto` in the Soot IR), and also the functions that performs comparisons (e.g., the string comparison APIs). We extract the constraints at each such sink.

Extracting the Path Constraints. Starting from the entry point (e.g., an event handler function f_i) of the ECG that eventually calls the network message sending APIs (e.g., function f_j), we iterate the Soot IR from the control flow path from f_i to f_j, perform the above taint analysis, and extract the path constraints at the encountered taint sinks. More specifically, if there is a taint source (based on the semantics of the IR), then we define the taint tag for the corresponding source (e.g., at line 7 in our running example in Figure 3 where `v0` is defined, we will define a taint tag, e.g., t_0, for `v0`); if there is a taint propagation, we propagate the taint as well; if there is a taint sink, then we extract the constraints based on the semantics of the `if-stmt` or the comparison APIs that use the tainted variables (e.g., at line 8, we will extract the taint tag t_0 with a constraint "t_0`.equalsIgnoreCase("")` not true").

Our taint analysis is inter-procedural. Whenever there is a function call, e.g., f_k, is called along the path from f_i to f_j, we will iterate the IR of f_k. If any of the arguments of f_k uses the tainted variables, or a global or heap variable is defined outside of f_k and it is also tainted (recall that Soot supports the def-use chain analysis), we will extract the path constraints concerning the return value computation if the return value of f_k is used as a path condition in the caller. In cases where a tainted variable (e.g., t_i) is defined outside f_k, we will find the definition of t_i, and solve the constraint for t_i if there is any.

4.3 Solving the Constraints

Having extracted the path constraints, we have to solve them and provide concrete proper input such that the final network sending APIs can be invoked. To this end, we use Z3-str [30] with the recent regular expression extension, an open source string solver based on Z3 [9], to solve the constraints we have collected. Interestingly, many of the string APIs used in Android has a corresponding Z3-str API (e.g., `length`, `contains`, `matches`, `startsWith`, `endsWith`). To make Z3 solver understand the extracted constraints, we have to translate them into the corresponding Z3-str APIs. After the API translation, the constraints can then be recognized by Z3-str, and we just invoke it to solve the constraints. This is a very standard procedure, and we just associate the resolved values for each constraint and store them in a configuration file. Later when the corresponding activities are executed, we load the configuration file and use these values to initialize the corresponding UI elements.

4.4 Runtime Instrumentation

Next, we have to really execute the app to concretely generate the network request messages with the proper input that has been resolved by our selective symbolic execution. The proper input has to be provided only when the activity is created (i.e., it is context-sensitive). Therefore, we have to instrument the app with our analysis code and execute them altogether.

State-of-the-art has three standard approaches to instrument the analysis code into the real app execution: (1) system code rewriting, (2) repackaging the app with analysis code, and (3) using the Android system tools (e.g., `adb`) . However, we found these approaches are inefficient for our large scale study. More specifically, if we rewrite the system code including the Android framework (as used by IntellDroid [29]), we have to execute the modified system code in an emulator. Similarly when using `adb` to supervise the

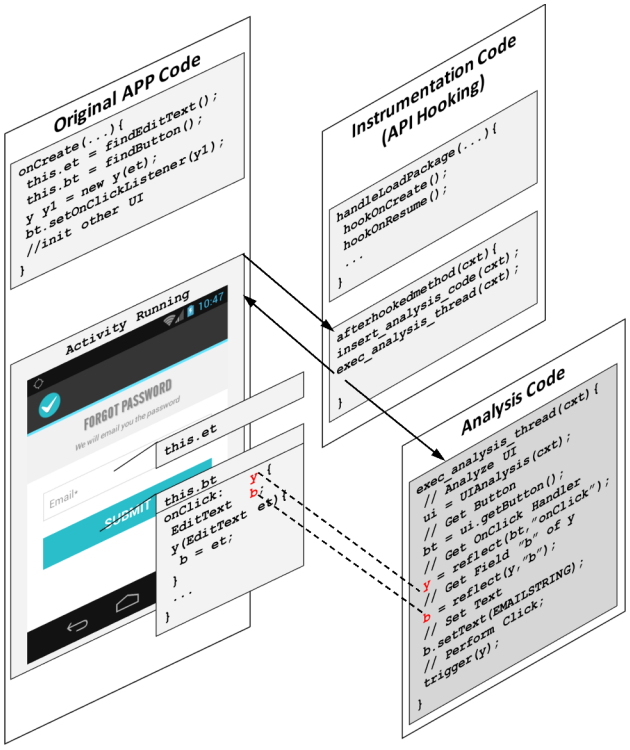

Figure 5: The Execution Flows between the Original App Code, the Instrumentation, and Our Analysis Code.

UI elements (as in DynoDroid [21]), it also requires to run in the emulator since the `ViewServer`, the critical component for recognizing each UI element by `adb`, is often turned off in real device. While repackaging seems to be the simplest, the repacked code may not be executed due to the integrity check by the app itself if there is any.

Therefore, to advance the sate-of-the-art, we propose a new dynamic instrumentation approach for Android app analysis running in real phones. The key idea is to use API hooking to intercept the control flow of an app execution, insert the analysis code and then invoke an analysis thread to perform an in-context analysis and provide the app with the solved proper input by using Java reflection [22]. Note that Java reflection provides a set of functions that allow us to examine or modify the classes, interfaces, fields and methods of an app at runtime in the Java virtual machine (VM). We can hence use this feature to access and manipulate the UI elements at runtime while the app is executing in the Dalvik VM.

How the analysis code is instrumented and executed in the app is illustrated in Figure 5. In particular, we divide all of the app executed code into three categories and describe how they will be executed below:

Original App Code. It still runs as normal in the main execution thread. Only when the events of our interest such as activity `onCreate` or `onResume` get executed, the control flow will transfer to the instrumentation code after the execution of the event handler, and this control flow transferring is achieved through the well-known API hooking technique [19].

Instrumentation Code. Once our API hooking intercepts an API of our interest (e.g., `onCreate`), it invokes the instrumentation code, which will still be executed in the app main execution thread.

Based on the current execution context, it injects the analysis code (via `insert_analysis_code`) into the analysis thread, and then starts the execution of the analysis thread. At this moment, there will be at least two execution threads: the main thread, and the analysis thread. The control flow of the main thread will still go back to the original app execution, and the analysis thread will start the execution after the instrumentation finishes.

Analysis Code. The analysis code performs an in-context analysis. As illustrated in Figure 5, our analysis code will first perform `UIAnalysis`, which is built atop Robotium [4], a library that allows us to retrieve the UI elements at runtime without any prior knowledge of the app. Based on the result from our `UIAnalysis`, then we get the UI elements of our interest such as the SUBMIT `button`. Next, it uses Java reflection to retrieve and manipulate the field and method associated with the corresponding UI elements. To choose which UI elements to manipulate is decided by our earlier selective symbolic execution. Specifically, our earlier analysis has determined which event handler we have to execute, such as the SUBMIT button `onClick`. Therefore, our analysis code will use the Java reflection to query the involved UI elements in `onClick` handler, and then setup their proper input values based on the constraints we have already solved, and invoke the `trigger` method to execute the `onClick` method in the analysis code.

4.5 Request Message Generation

With all the building blocks enabled, we next describe how we navigate various app activities and finally generate the desired request messages to expose the URLs. An Android app can have many activities, and they will only be executed when the corresponding events are generated. The first activity will be executed automatically when the app is started. When a given activity is executed, we hook its `onCreate` event, and then we are able to inspect all the UI element of the current activity. Based on the UI element, we are able to determine all other activities that can be invoked from the current activity. As described earlier, we use a DFS strategy to execute the activity. A request message will be generated after the corresponding network message sending API is executed. We then log this message and extract its URLs if there is any as our final output.

5. EVALUATION

We have implemented SMARTGEN atop Android 4.2 platform. In particular, we implement our ECG construction using the Soot [2] framework and taint analysis using FlowDroid [11]. We use Z3-Str [30] to solve the path constraints and implement our dynamic runtime instrumentation with the v2.7 Xposed [8] framework. In total, we wrote about $7,000$ lines of Java code and $1,000$ lines of Python code atop these open source frameworks.

In this section, we present the evaluation result of SMARTGEN. During the evaluation, we aim to answer the following questions as we have set up in our design objectives in §2.1:

- **How Automated?** Can SMARTGEN be executed without any human intervention?
- **How Scalable?** Can SMARTGEN handle a large-volume dataset? How fast is SMARTGEN in processing each app?
- **How Systematic?** Does the selective symbolic execution in SMARTGEN really encounter many constraints? How significant is this component in terms of the contribution to the number of messages generated and URLs exposed?

Table 1: Summary of the Performance of SMARTGEN.

Item	Value
# Apps	$5,000$
Size of the Dataset (G-bytes)	126.2
Time of the first two phases analyses (s)	$90,143$
# Targeted API Calls	$147,327$
# Constraints	$47,602$
# UI Configuration files generated	$25,030$
Time of Dynamic Analysis (s)	$486,446$
# Request Messages	$257,755$
# Exposed URLs	$297,780$
# Unique Domains	$18,193$
Logged Message Size (G-bytes)	24.0

We first describe how we set up the experiment in §5.1, and then present the detailed result in §5.2. Finally, we present how to use the exposed URLs in a harmful URL detection study in §5.3.

5.1 Experiment Setup

Dataset Collection. The inputs to SMARTGEN are Android apps. Since there are millions of Android apps, we cannot test all of them. We thus decided to crawl the top $5,000$ (in terms of number of installs) ranked apps from the Google Play. To this end, we first used the Scrapy framework [5] to crawl all meta-data of an app (including the number of installs and the category) from Google Play, which contained the meta-data for over 1.5 million apps as of March 2016. Then we ranked the apps based on the number of installs, and crawled the APK of each app starting from the top-ranked one. After downloading an app, we checked whether it has the Internet permission by looking at its `AndroidManifest.xml` file. If it did not have any Internet permission, we excluded this app from our data set. We kept crawling until our data set reached $5,000$ apps. During the crawling, we discarded 219 apps that did not have the Internet permission in their manifest files, such as Shake Calc (a calculator) and aCalendar (an Android calendar), as well as 473 apps that either do not run in the ARM architecture (apps utilizing x86) or that could not be downloaded because of the region restrictions imposed by Google Play. The last app we downloaded had over one million installs.

Environment Configuration. SMARTGEN contains three phases of analyses: static analysis, selective symbolic execution, and dynamic analysis. The first two phases were executed in an Ubuntu 14.04 server machine running an Intel Xeon CPUs with 16 cores and 2.393GHz and 24GB memory. The last phase was executed in a Samsung Galaxy S III phone running Android 4.2. To make the static analysis and symbolic execution run faster, we started 10 threads to perform these analyses in our server. After the analysis finished, each target app was then automatically pushed to the Galaxy phone, along with the solved constraints, which are stored in the format as configuration files. The finally generated request messages for each app were collected in a man-in-the-middle proxy. To gather the HTTPS messages, we installed a root certificate in the phone.

5.2 Detailed Result

Overall Performance. The overall statistics on how each phase of our analysis was performed is presented in Table 1. We can see that these $5,000$ apps consumed 126.2 GB disk space. The first two phases of our analysis (i.e., static analysis and symbolic execution) took 90,143 seconds (i.e., 25.04 hours) in total, and each app took 18.03 seconds on average. During this analysis, it identified $147,327$ calls to the targeted APIs, extracted $47,602$ constraints,

Table 2: Statistics of the Extracted String Constraints

Constraints Name	# Constraints
Not null	25,855
String_length	13,858
String_isEmpty	377
String_contains	196
String_contentEquals	43
String_equals	3,087
String_equalsIgnoreCase	991
String_matches	448
String_endsWith	11
String_startsWith	64
TextUtils_isEmpty	2,355
Matcher_matches	317

and generated 25,030 UI configuration files based on the solved constraints.

Meanwhile, the details of the extracted string constraints are presented in Table 2. (Note that we also encountered other integer constraints, such as when a value needs to be greater than 18; the details of these constraints are not presented here). We notice that, interestingly, there are many "Not `null`" constraints. This is actually because during an app execution, `NullPointer` may cause crashes and developers (or the system code) thus check it very often. Meanwhile, to validate whether a UI item contains a user input, we noticed developers also often use a `String_length` constraint (to make sure it is not 0). Some apps also use `String_length` to validate phone number input. Also, we found some apps just use `String_contains` with "`@`" to validate an email address input, and some other apps use sophisticated regular expression (e.g., `Matcher_matches`) for the matching.

With the solved constraints, we then performed dynamic analysis on each app in our Galaxy phone. In total, it took 486,446 seconds (i.e., 135 hours) to execute these 5,000 apps (each app needed 97 seconds on average). Note that among the 97 seconds, the installation and uninstallation time is on average 17 seconds. However, if we execute an app inside an emulator, the installation time for an app with 25M will take about 60 seconds. That is one of the reasons why we designed SMARTGEN to use a real phone. During the dynamic analysis, we observed 257,755 request messages (55.7% uses HTTP protocol, and 44.3% uses HTTPS) generated by the tested apps, and in total 297,780 URLs in both request and response messages. Among them, there are 18,193 unique domains in these URLs. The final size for all the traced request and response messages collected at our proxy is 24.0 GB.

Comparison with Monkey. To understand the contribution of our selective symbolic execution, we compare SMARTGEN with a widely used dynamic analysis tool Monkey [7]. At a high level, Monkey is a program, executed in an emulator or a real phone, which can generate pseudo-random streams of user events, such as clicks, touches, or gestures, as well as a number of system-level events, all for app testing. For a fair comparison, we also run Monkey in our real Galaxy phone to test each of our app, and configure Monkey to generate 2,000 events under the time interval of 100 milliseconds. That is, for each app, Monkey will take up to 200 seconds to just test it.

In total it took 1,083,530 seconds (i.e., 301 hours) to process these apps. Each app took on average 216.7 seconds (among them around 200 seconds for the testing, and 17 seconds for the installation and uninstallation). We have to also note that it is *not* 100% automated while using Monkey for the testing. This is because Monkey randomly sends events to the system without specifying the recipients. These random inputs may click system buttons, which may lock the screen, turn off the network connection, and

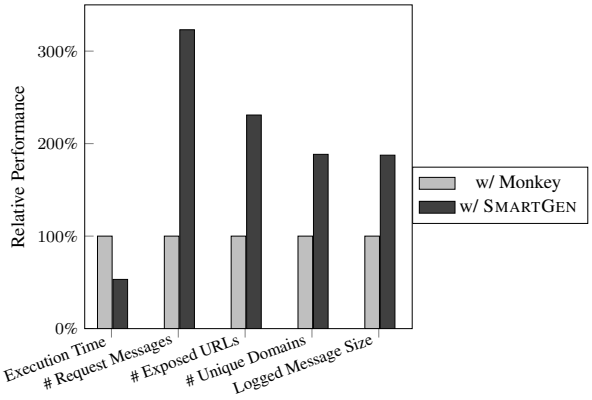

Figure 6: Comparison between SMARTGEN and Monkey.

even shutdown the phone. Therefore, we disabled the screen locking functionality, and also developed a daemon program to constantly check the Internet connectivity and turn on the networking if necessary, but we cannot disable the phone power off event and must manually power on the phone. This is the only event Monkey cannot handle automatically and we encountered 17 phone power off events. We excluded the power-off and restart time in our evaluation in this case. For all these tested apps, with Monkey they generated 79,778 request messages, with 6,384 domain names. The total size of the logged message is 12.8 GB.

A detailed comparison between SMARTGEN and Monkey for these tested apps is presented in Fig. 6. We compare them based on their execution time, the total number of request messages generated, the total number of domains in the requested message, and finally the total size of the request message. We can see that SMARTGEN only took 53%, i.e., $(90,143 + 486,446)/1,083,530$, of the execution time of Monkey, but it generates 3.2X request messages, 2.3X unique URLs, 1.9X unique domains, and 1.9X logged message size, compared to the result from Monkey.

5.3 Harmful URL Detection

Having so many URLs from the top 5,000 mobile apps, we are then interested in whether there is any harmful URLs. To this end, we submitted all of the exposed 297,780 URLs to harmful URL detection service at VirusTotal, which then further identified 8,634 unique harmful URLs. Note that VirusTotal has integrated 68 malicious URL scanners (as time of this experiment), and each submitted brand new URL is analyzed by all of the scanners. The scanners that have identified at least one harmful URLs are reported in the first column of Table 3, followed by the number of Phising sites, the number of malware (i.e., the URL is identified as malware), and the number of malicious sites from the 2nd to the 4th columns, respectively. The last column reports the total number of harmful URLs identified by the corresponding scanners, and the last row reports the number of unique URLs in each category. The total number of unique malicious URL is 8,634 because there are 387 sites being detected both malware and malicious. Also, note that one harmful URL can be identified by several engines. That is, there are some overlapped URLs in the last column of Table 3. To clearly show those overlaps, we present the number of harmful URLs and the number of engines that recognize those harmful URLs in Table 4. Interestingly, we can see that most harmful URLs are detected by just one of the engines, and only one URL is detected simultaneously by 8 engines. Based on the timestamp of the queried result from VirusTotal, we notice that 83% of the URLs are the first time

Table 3: Statistics of Harmful URLs Detected by Each Engine

Detection Engine	#Phishing Sites	#Malware	#Malicious Sites	Σ# URLs
ADMINUSLabs	0	0	4	4
AegisLab WebGuard	0	0	1	1
AutoShun	0	0	863	863
Avira	2062	941	0	3003
BitDefender	0	191	0	191
Blueliv	0	0	5	5
CLEAN MX	0	0	14	14
CRDF	0	0	150	150
CloudStat	0	0	1	1
Dr.Web	0	0	2330	2330
ESET	0	75	0	75
Emsisoft	1	43	0	44
Fortinet	8	469	0	477
Google Safebrowsing	0	13	2	15
Kaspersky	0	2	0	2
Malwarebytes hpHosts	0	1103	0	1103
ParetoLogic	0	800	0	800
Quick Heal	0	0	2	2
Quttera	0	0	6	6
SCUMWARE.org	0	8	0	8
Sophos	0	0	56	56
Sucuri SiteCheck	0	0	248	248
ThreatHive	0	0	8	8
Trustwave	0	0	80	80
Websense ThreatSeeker	0	0	56	56
Yandex Safebrowsing	0	173	0	173
Σ#Harmful URLs	2071	3818	3826	9715
Σ#Unique Harmful URLs	2071	3722	3228	8634

Table 4: # Engines of Harmful URLs

Detected by # Engines	# Unique Harmful URLs
8	1
7	1
6	2
5	13
4	63
3	33
2	751
1	7770
Σ Unique Harmful URLs	8634

analyzed by VirusTotal. Among the detected $8,634$ URLs, we also notice that 84% of them are new harmful URLs (because of our research).

While we could just trust the detection result from VirusTotal, to confirm indeed these URLs are malicious we manually examined the one that has been identified by 8 engines. Interestingly, this URL actually points to an APK file. We then visited this URL and downloaded the APK. We also submitted this suspicious APK file to VirusTotal, and this time, 14 out of 55 file scanners reported that this APK is malicious. We reverse engineered this file and found it tried to acquire the root privilege of the phone by exploiting the kernel vulnerabilities, which undoubtedly proved it is a harmful URL.

6. LIMITATIONS AND FUTURE WORK

SMARTGEN clearly has limitations. First, there might be some missing path in ECG (if an edge is missed by EdgeMiner [16]), or infeasible paths that cannot be solved (currently our solver terminates if it cannot provide any result after 300 seconds). Second, not all of the app activities have been explored, especially if there is an access control in the app. More specifically, certain app activities are only displayed if the user has successfully logged in. However, SMARTGEN did not perform any automatic registration with these $5,000$ apps, and it is certainly not able to trigger these activities. Therefore, how to trigger these activities for a given mobile app is one of our immediate future works.

Currently, we only demonstrated how to use the exposed URLs to detect whether an app communicates with any malicious sites. There are certainly many other applications such as server vulnerability identification [31]. For instance, we can use the generated server request messages as seeds to perform the penetration testing to see whether the server contains any exploitable vulnerabilities such as SQL injection, cross-site-scripting (XSS), cross-site request forgery (CSRF), etc. We leave the study of the vulnerability fuzzing to our another future work.

We can also apply the selective symbolic execution of SMARTGEN to solve other problems. For instance, by changing the targeted APIs to those security-sensitive ones (e.g., `getDeviceId`), we can collect and solve the constraints along the execution path to trigger these APIs. Through this, we are likely able to further observe how sensitive data is collected and perhaps find privacy leakage vulnerabilities in real apps. Part of our future work will also explore these applications.

7. CONCLUSION

We have presented SMARTGEN, a tool to automatically generate server request messages and expose the server URLs from a mobile app by using selective symbolic execution, and demonstrated how to use SMARTGEN to detect malicious sites based on the exposed URLs for the top $5,000$ Android apps in Google Play. Unlike prior efforts, SMARTGEN focuses on the constraints from the UI elements and solves them to trigger the networking APIs. Built atop API hooking and Java reflection, it also features a new runtime app instrumentation technique that is able to more efficiently instrument an app and perform an in-context analysis. Our evaluation with the top $5,000$ ranked mobile apps have demonstrated that with SMARTGEN we are able to find $297,780$ URLs, and among them actually $8,634$ are malicious sites according to the URL classification result from VirusTotal.

Acknowledgment

We thank VirusTotal for providing premium services during the evaluation of large volume of (new) URLs. We also thank the anonymous reviewers for their helpful comments. This research was supported in part by AFOSR under grant FA9550-14-1-0119 and FA9550-14-1-0173, NSF award 1453011. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and NSF.

8. REFERENCES

[1] Automatic sql injection and database takeover tool. http://sqlmap.org/.

[2] A framework for analyzing and transforming java and android apps. https://sable.github.io/soot/.

[3] Owasp fiddler addons for security testing project. https://www.owasp.org/index.php/OWASP_Fiddler_Addons_for_Security_Testing_Project.

[4] Robotium: User scenario testing for android. https://github.com/RobotiumTech/robotium.

[5] Scrapy, a fast high-level web crawling & scraping framework for python. https://github.com/scrapy/scrapy.

[6] Statistics and facts about mobile app usage. http://www.statista.com/topics/1002/mobile-app-usage/.

[7] Ui/application exerciser monkey. https://developer.android.com/tools/help/monkey.html.

[8] Xposed module repository. http://repo.xposed.info/.

[9] The z3 theorem prover. https://github.com/Z3Prover/z3.

[10] 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*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[11] 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*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[12] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[13] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[14] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, 2014.

[15] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[16] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[17] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[18] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[19] G. Hunt and D. Brubacher. Detours: Binaryinterception ofwin 3 2 functions. In *3rd USENIX Windows NT Symposium*, 1999.

[20] B. Liu, S. Nath, R. Govindan, and J. Liu. Decaf: Detecting and characterizing ad fraud in mobile apps. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 57–70, Berkeley, CA, USA, 2014. USENIX Association.

[21] 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*, pages 224–234. ACM, 2013.

[22] G. McCluskey. Using java reflection. *Java Developer Connection*, 1998.

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

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

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

[26] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

[27] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 190–203. ACM, 2014.

[28] 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 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, February 2014.

[29] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

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

[31] C. Zuo, W. Wang, R. Wang, and Z. Lin. Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.