6-2021

When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid

Daoyuan WU

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

Rocky CHANG

Citation

WU, Daoyuan; GAO, Debin; DENG, Robert H.; and CHANG, Rocky. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid. (2021). *Proceedings of the 51st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*. 543-554. Research Collection School Of Computing and Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/6540

2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) | 978-1-6654-3572-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN48987.2021.00063

When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid

Daoyuan Wu
Department of Information Engineering
The Chinese University of Hong Kong
Hong Kong SAR, China
dywu@ie.cuhk.edu.hk

Debin Gao, Robert H. Deng
School of Information Systems
Singapore Management University
Singapore, Singapore
{dbgao, robertdeng}@smu.edu.sg

Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
csrchang@comp.polyu.edu.hk

Abstract*—Widely-used Android static program analysis tools, e.g., Amandroid and FlowDroid, perform the *whole-app* inter-procedural analysis that is comprehensive but fundamentally difficult to handle modern (large) apps. The average app size has increased three to four times over five years. In this paper, we explore a new paradigm of *targeted* inter-procedural analysis that can skip irrelevant code and focus only on the flows of security-sensitive sink APIs. To this end, we propose a technique called *on-the-fly bytecode search*, which searches the disassembled app bytecode text just in time when a caller needs to be located. In this way, it guides targeted (and backward) inter-procedural analysis step by step until reaching entry points, without relying on a whole-app graph. Such search-based inter-procedural analysis, however, is challenging due to Java polymorphism, callbacks, asynchronous flows, static initializers, and inter-component communication in Android apps. We overcome these unique obstacles in our context by proposing a set of bytecode search mechanisms that utilize flexible searches and forward object taint analysis. Atop this new inter-procedural analysis, we further adjust the traditional backward slicing and forward constant propagation to provide the complete dataflow tracking of sink API calls. We have implemented a prototype called BackDroid and compared it with Amandroid in analyzing 3,178 modern popular apps for crypto and SSL misconfigurations. The evaluation shows that for such sink-based problems, BackDroid is 37 times faster (2.13 v.s. 78.15 minutes) and has no timed-out failure (v.s. 35% in Amandroid) while maintaining close or even better detection effectiveness.

I. Introduction

Static analysis is a common program analysis technique extensively used in the software security field. Among existing Android static analysis tools, Amandroid [39], [40] and FlowDroid [8], [10] are the two most advanced and widely used ones, with the state-of-the-art dataflow analysis capability. Both perform the *whole-app* inter-procedural analysis that starts from all entry points and ends in all reachable code nodes. Such analysis is comprehensive but ignores specific analysis requirements, and often comes at the cost of huge overheads. For example, a dataflow mining study [11] based on FlowDroid used a compute server with 730 GB of RAM and 64 CPU cores. Even with such a configuration, "*the server sometimes used all its memory, running on all cores for more than 24 hours to analyze one single Android app*" [11].

As a result, small apps were often used in prior studies. For example, AppContext [44] selected apps under 5MB for analysis because they found that not enough apps could be successfully analyzed by its underlying FlowDroid tool. Although HSOMiner [34] increased the size of the analyzed apps, the average app size is still only 8.4MB. Even for these small apps, AppContext timed out for 16.1% of the 1,002 apps tested, and HSOMiner similarly failed in 8.4% of 3,000 apps, causing a relatively high failure rate. Hence, this is not only a performance issue but also the detection burden. Additionally, third-party libraries were often ignored. For example, Amandroid by default skipped the analysis of 139 popular libraries, such as AdMob, Flurry, and Facebook.

However, the size of modern apps keeps on increasing. According to our measurement, the average and median sizes of popular apps on Google Play have increased three and four times, respectively, over a period of five years. With these modern apps, we re-evaluate the cost of generating a relatively precise whole-app call graph using the latest FlowDroid, and find that even this task alone (i.e., without conducting the subsequently more expensive dataflow analysis) could be sometimes expensive — 24% of apps failed even after running for 5 hours each. Hence, a scalable Android static analysis is needed to keep pace with the upscaling trend in modern apps. Fortunately, security studies are usually interested only in a small portion of code that involves the flows of security-sensitive sink APIs. For example, Android malware detection [48] is mostly interested in the sink APIs that can introduce security harms (e.g., `sendTextMessage()`), and vulnerability analysis often spots particular patterns of the code [17]. Therefore, it is possible for security-oriented tools to perform a targeted analysis from the selected sinks.

In this paper, we explore a new paradigm of *targeted* (v.s. the traditional whole-app) inter-procedural analysis that can skip irrelevant code and focus only on the flows of security-sensitive sink APIs. To achieve this goal, we propose a technique called *on-the-fly bytecode search*, which searches the disassembled app bytecode text just in time when a caller needs to be located so that it can guide targeted (and backward) inter-procedural analysis step by step until reaching entry points. We combine this technique with the traditional program analysis and develop a static analysis tool called BackDroid, for the efficient and effective targeted security vetting of *mod-*

978-1-6654-3572-7/21/$31.00 ©2021 IEEE
DOI 10.1109/DSN48987.2021.00063

ern Android apps. Since the whole-app analysis is no longer needed in BackDroid, the required CPU and memory resources are controllable regardless of app size. Such a design, however, requires us to solve several unique challenges. Specifically, it is challenging to perform effective bytecode search over Java polymorphism (e.g., superclasses and interfaces), callbacks (e.g., `onClick()`), asynchronous flows (e.g., `AsyncTask.execute()`), static initializers (i.e., static `<clinit>()` methods), and ICC (inter-component communication) in Android apps. Note that these obstacles are different from when they appeared previously [10], [39], where the challenge was to determine object types instead of hindering the searches.

To overcome those obstacles in our context, we propose a set of bytecode search mechanisms that utilize flexible searches and object taint analysis. First, we present a method signature based search that constructs appropriate search signatures to directly locate callers for static, private, and constructor callee methods. This basic search also works well for child classes by just launching one more signature search with the child class. Second, for complex situations with superclasses, interfaces, callbacks, and asynchronous flows, we propose an advanced search mechanism because directly searching callers' signatures in these situations would hit nothing. Instead, we first search the callee class's object constructor(s) that can be accurately located, and from those constructors, we perform forward object taint analysis until reaching caller sites. Third, we further propose several special search mechanisms, including (i) a recursive search to determine the reachability of static initializers; (ii) a two-time ICC search that first searches for both ICC calls and parameters and then merges their search results; and (iii) an on-demand search for Android lifecycle handlers, e.g., `onStart()` and `onResume()` methods.

Atop our new paradigm of inter-procedural analysis, we further adjust the traditional backward slicing and forward constant propagation to provide the dataflow tracking of sink API calls. Specifically, we first generate a structure called *self-contained slicing graph* (SSG) to record all the slicing information and inter-procedural relationships during the search-based backtracking. To faithfully construct an SSG, we not only reserve the raw typed bytecode statements but also taint across fields and arrays, as well as add off-path static initializers on demand. On top of the generated SSGs, we then conduct forward constant and points-to propagation over each SSG node to propagate and calculate the complete dataflow representation of target sink API parameters.

To evaluate BackDroid's efficiency and efficacy, we compare it with the state-of-the-art Amandroid [39], [40] tool in analyzing modern apps for crypto and SSL misconfigurations, two common and serious sink-based problems that were also tested by Amandroid in [40]. Specifically, we first select a set of 3,178 modern apps with over one million installs each and were updated in recent years as our basic dataset. We then pre-search them to obtain 144 apps with all the relevant sink APIs so that Amandroid would not waste its analysis even without the bytecode search capability. The average and median sizes of these apps are 41.5MB and 36.2MB, respectively. We use a default parameter configuration of Amandroid and run both tools on a machine with 8-core CPU and 16GB memory, a memory configuration often used in many previous studies (e.g., [34], [36], [44], [46]). Moreover, we give Amandroid sufficient running time with a large timeout of five hours (or 300 minutes) per app (the timeout setting explicitly reported in prior studies [9], [15], [34], [44] was 30, 60, and 80 minutes).

Our evaluation shows that BackDroid achieves much better performance while its detection effectiveness is close to, or even better than, that of Amandroid. First, BackDroid's overall performance is 37 times faster than that in Amandroid, requiring only 2.13min (or minutes) for the median analysis time while that in Amandroid is 78.15min. Indeed, BackDroid can quickly analyze one-third of the apps within one minute each (v.s. 0% in Amandroid), and 77% of apps can be finished within 10 minutes each (v.s. 17% in Amandroid). Moreover, BackDroid has no timed-out failure and only three apps exceeding 30min. In contrast, the timed-out failure rate in Amandroid is as high as 35%. On the other hand, BackDroid still maintains close detection effectiveness for the 30 vulnerable apps detected by Amandroid: BackDroid uncovered 22 of 24 true positives and avoided six false positives. Furthermore, BackDroid discovered 54 additional apps with potentially insecure ECB and SSL issues. One half were the timed-out failures, but the rest were due to the skipped libraries, unrobust handling of asynchronous flows/callbacks, and occasional errors in Amandroid's whole-app analysis.

Availability: To allow more usages of BackDroid and facilitate future research, we have released the source code of BackDroid at https://github.com/VPRLab/BackDroid.

II. Background and Motivation

A. Related Work of Android Static Analysis

Different from the classical program analysis (e.g., for Java and C/C++) that handles only one entry point, static analysis of Android apps needs to consider many entry points that are implicitly called by the Android framework. These entry points are the lifecycle handler methods (e.g., `onCreate()`) of different components registered in an app configuration file called `AndroidManifest.xml` (or manifest thereafter). An important task of Android static security analysis is to test the reachability from entry points to security-sensitive sink API calls. Since such control-flow reachability usually involves multiple Java methods (or procedures, formally), an *inter*-procedural, instead of *intra*-procedural, analysis is a must in Android static analysis. Otherwise, we cannot determine whether a sink API call is valid, i.e., not dead code or uninvoked libraries (both are common in Android apps).

Existing Android static tools need to launch the *whole-app* analysis for inter-procedural analysis. Specifically, before they can perform any inter-procedural dataflow taint [20] or backward slicing [16] analysis, they all generate certain kinds of whole-app graphs [47], [50]. Some are constructed systematically, such as a points-to graph for all objects in Amandroid [39], a lifecycle-aware call graph in FlowDroid [10], a system dependency graph in R-Droid [12], and a context-sensitive call graph in CHEX [28], DroidSafe [21], and IntelliDroid [41]. Among them, Amandroid's whole-app graph, generated along with the dataflow analysis, is the most precise one. Others are built in an ad-hoc manner and are thus less accurate, e.g., a class hierarchy based control flow graph [15], [22], [49], a method signature based call graph [24], and an *intra*-procedural type-inference based call graph [13].

However, *whole-app* Android static analysis inherently has scalability difficulty, which was also realized by some recent works [13], [33], [34]. For example, because of the worry that it is slow to launch complicated code analysis in a set of apps with an average size of 8.43MB, HSOMiner [34] proposed to combine lightweight program analysis with machine learning for the large-scale detection of hidden sensitive operations. Similarly, in the work by Bianchi et al. [13], it stated the challenge in analyzing recent real-world Android apps due to their large amount of code. As a result, they performed an *intra*-procedural type-inference analysis to construct their whole-app call graph. More recently, to speed up the cryptographic vulnerability detection for massive-sized Java projects, CryptoGuard [35], proposed the crypto-specific slicing that uses a set of domain knowledge on top of Soot's intra-procedural dataflow analysis to balance the precision and runtime. To the best of our knowledge, BackDroid is the first tool that does not perform a whole-app analysis *yet* still offers effective inter-procedural analysis.

To build Android static analyses, some common technical tools are often used. Notably, the Soot [25] and WALA [6] analysis frameworks have been used in many prior works. They provide the underlying intermediate representation (IR), call graph generation, and some basic support of data flow analysis. In BackDroid, we also leverage Soot's Shimple IR (an IR in the Static Single Assignment form) to build our own dataflow analysis but use bytecode search, instead of Soot's call graph, to support the inter-procedural analysis.

B. The Upscaling Trend of Modern App Sizes

Both Amandroid and FlowDroid were initially proposed in 2014. Although they are still being maintained and improved over these years, handling large apps was not considered a design objective in the first place. However, as we will show in this subsection, modern popular apps have increased their sizes dramatically over a period of five years from 2014 to 2018. This is not too surprising, considering that 1,448 of the top 10,713 apps studied in 2014 [31] had already been updated on a bi-weekly basis or even more frequently. Note that although app size increase is not fully due to code change (also related to user interface XML code and resource change), it is the most common metric for describing an app dataset and allows a comparison between our and prior datasets (e.g., [34], [44]).

To measure the changes in the app sizes, we first obtain a set of modern apps. Specifically, we collected a set of 22,687 popular apps on Google Play in November 2018 by correlating the AndroZoo repository [7] with the top app lists available on https://www.androidrank.org. Each app in this set had at least one million installs on Google Play. We then recorded the app sizes and DEX file dates (if any).

TABLE I: A summary of average and median app sizes over a period of five years from 2014 to 2018.

Year	Average Size	Median Size	# Samples
2014	13.8MB	8.4MB	2,840
2015	18.8MB	12.4MB	1,375
2016	21.6MB	16.2MB	3,510
2017	32.9MB	30.0MB	1,706
2018	42.6MB	38.0MB	3,178

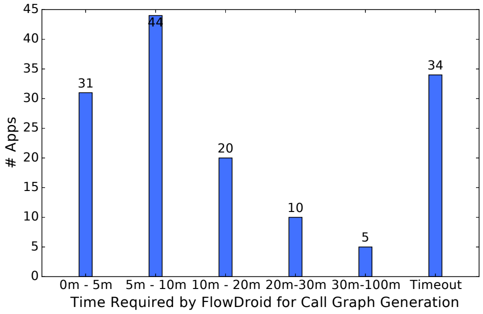


Fig. 1: FlowDroid's call graph generation time for a set of 144 modern apps (under a timeout of 5 hours each).

Table I summaries the average and median app sizes over a period of five years from 2014 to 2018. We can see that in 2014, the average and median app size is only 13.8MB and 8.4MB, respectively. This number almost doubles in 2016, with an average size of 21.6MB and a median size of 16.2MB. It further doubles after two years, with an average app size of 42.6MB in 2018. This clearly shows that modern apps have dramatically increased their app sizes, and they are expected to further enlarge as more functionalities are added.

C. The Cost of Generating a Whole-app Call Graph

With modern apps, we now re-evaluate the cost of generating a relatively precise whole-app call graph. Specifically, we measure the execution time required by FlowDroid 2.7.1[1] to analyze a set of 144 modern apps with the average size of 41.5MB, under the same hardware configuration for our experiments of BackDroid and Amandroid in §VI. We choose FlowDroid because it decouples the logic of call graph generation and dataflow taint analysis (whereas Amandroid cannot), which allows us to measure the cost of generating call graphs only. To increase the precision of FlowDroid's call graph generation, we use the context-sensitive geomPTA [3] call graph algorithm, instead of the context-insensitive SPARK algorithm [2]. However, we do not launch IccTA [27] to further transform the generated call graphs with inter-component edges. This sacrifices certain accuracy but keeps stability.

We find that even generating a whole-app call graph alone (without performing the subsequently more expensive dataflow analysis) could be sometimes expensive. Figure 1 presents FlowDroid's call graph generation time for a set of 144 modern apps. Although the call graphs of 31 (21.5%) apps could be generated within five minutes, the median time of call

[1] The latest release at the time of our evaluation from 2019 to 2020. Note that this version of FlowDroid has continued improving the performance on top of the most recently published one in 2016 [4], [8] for over two years.

graph generation in FlowDroid is still around 10 minutes (9.76min) per app. Considering the total analysis time required by BackDroid for the same set of apps is only 2.13min (see §VI), such a call graph generation is already 4.58 times slower. Besides the performance concern, the detection burden caused by timed-out failures is much more serious. We can see that as high as 24% of apps failed even after running for 5 hours each, causing no result outputted for these 34 modern apps.

III. Overview and Challenges

Motivated by the incapability of *whole-app* inter-procedural analysis to analyze modern Android apps, we explore a new paradigm of *targeted* inter-procedural analysis that can skip irrelevant code and focus only on the flows of security-sensitive sink APIs. This new paradigm is enabled by a technique called *on-the-fly bytecode search*, which searches the disassembled app bytecode text just in time to guide targeted (and backward) inter-procedural analysis until reaching entry points. We implement this technique into a tool called BackDroid, and further adjust the traditional backward slicing and forward constant propagation to provide the dataflow tracking of targeted sink APIs. Figure 2 presents a high-level overview of BackDroid, which works in the following steps:

1) *Preprocessing by disassembling bytecode into plaintext.* Given an Android app(s), BackDroid first extracts its original bytecode and manifest files. After that, it not only transforms bytecode into a suitable intermediate representation (IR) as in typical Android analysis tools but also employs `dexdump` [1] to disassemble (merged, if multidex [5] is used) bytecode to a plaintext.
2) *Bytecode search for targeted inter-procedural analysis.* With the disassembled bytecode plaintext, BackDroid immediately locates the target sink API calls by performing a text search of bytecode plaintext (in the bytecode search space) and initiates the analysis from there (in the program analysis space). To support the inter-procedural analysis with no call graph, BackDroid performs on-the-fly bytecode search to identify caller methods on demand.
3) *Performing the adjusted backward slicing to generate SSG.* To also provide the dataflow tracking atop our search-based inter-procedural analysis, BackDroid performs the traditional backward slicing but adjusts it into our new context. Specifically, during the inter-procedural backtracking, BackDroid generates a self-contained slicing graph (SSG) for each sink API call to record all the slicing information and inter-procedural relationships resolved by the search.
4) *Forward analysis over SSG to propagate and calculate dataflow.* Atop the generated SSGs, BackDroid launches the classical forward constant propagation [38] to propagate and calculate dataflow facts from entry points to sink APIs. It also propagates object points-to [26] information to remove potential ambiguity. Eventually, with the support of SSG, BackDroid is able to output the *complete* dataflow representation (either a constant or expression) of target sink API parameters.

Challenges. Since BackDroid is the first inter-procedural analysis tool without relying on a whole-app graph, its major novelty and the biggest challenge is how to perform on-the-fly bytecode search to locate caller methods in a backward manner. This is difficult because of Java polymorphism (e.g., superclasses and interfaces), callbacks, asynchronous Java/Android flows, static initializers, and inter-component communication, all of which make a basic signature-based search infeasible. To address this challenge, we propose a novel bytecode search technique in §IV. Another challenge is how to adjust the traditional backward slicing and forward analysis into our new paradigm of targeted inter-procedural analysis, the solutions of which will be presented in §V.

IV. On-the-fly Bytecode Search

In this section, we present our novel bytecode search technique to locate caller methods on the fly, which is the key to BackDroid's targeted inter-procedural analysis. We first present the basic signature-based search in §IV-A and an advanced search mechanism with forward object taint analysis in §IV-B. We then elaborate on three special search mechanisms from §IV-C to §IV-E to effectively search over static initializers, Android ICC (inter-component communication), and Android lifecycle handlers (e.g., `onStart`/`onResume`).

While different searches are separated in different subsections, BackDroid's design is systematic (i.e., the search process is not ad-hoc and does not bind to specific problems). However, BackDroid's search process involves some tedious details about bytecode-level method representations. Notably, Java methods could be classified into two kinds at the bytecode level: (i) the method has only one possible representation (i.e., the method's class name is the current class); and (ii) the method may have multiple representations, i.e., the method's class name could be the current/parent/interface class. For interface methods, the method name itself could even be changed. For each callee method, BackDroid first determines its nature and dispatches it to corresponding search handlers.

A. Basic Search by Constructing Search Signature(s)

To illustrate our search process, we use a real popular app, LG TV Plus[2], which has over 10 million installs on Google Play, as a running example. In Figure 3, we have already used initial bytecode search to find a target method (i.e., the one with a sink API call), `<com.connectsdk.service.netcast.NetcastHttpServer: void start()>`. For inter-procedural analysis, our next step is to uncover its caller method (i.e., `<com.connectsdk.service.NetcastTVService$1: void run()>`) and its call site (i.e., statement `virtualinvoke $r13.<com.connectsdk.service.netcast.NetcastHttpServer: void start()>()`). As we will explain, searching its caller can be done directly with the following (method) signature-based search, because the target callee method here is a `private` Java method.

[2] https://play.google.com/store/apps/details?id=com.lge.app1

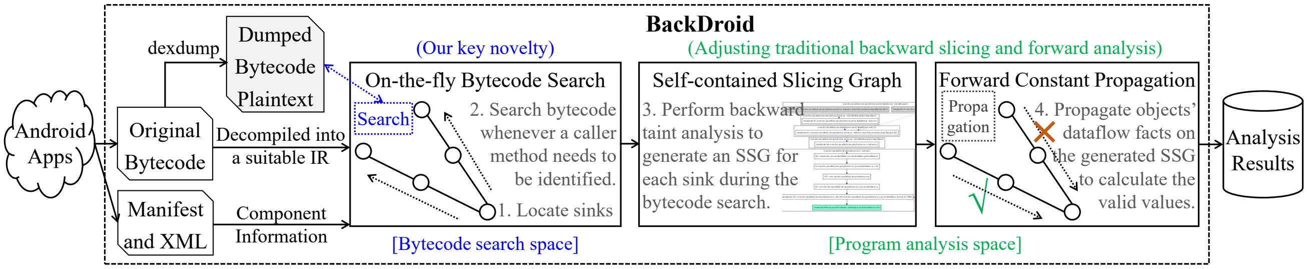
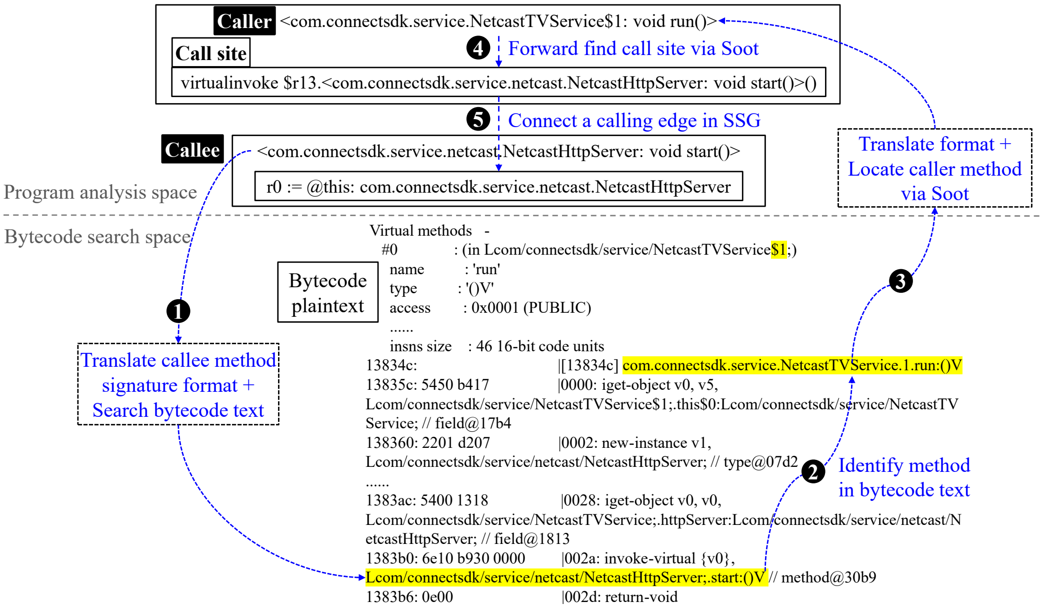

BackDroid
Android Apps
dexdump
Dumped Bytecode Plaintext
Original Bytecode
Decompiled into a suitable IR
Manifest and XML
Component Information
(Our key novelty)
On-the-fly Bytecode Search
Search
2. Search bytecode whenever a caller method needs to be identified.
1. Locate sinks
[Bytecode search space]
(Adjusting traditional backward slicing and forward analysis)
Self-contained Slicing Graph
3. Perform backward taint analysis to generate an SSG for each sink during the bytecode search.
Forward Constant Propagation
Propa gation
4. Propagate objects' dataflow facts on the generated SSG to calculate the valid values.
√
[Program analysis space]
Analysis Results

Fig. 2: A high-level overview of BackDroid.


Caller
Call site
4 Forward find call site via Soot
5 Connect a calling edge in SSG
Callee
Program analysis space
Bytecode search space
Translate format + Locate caller method via Soot
3
Bytecode plaintext
1
Translate callee method signature format + Search bytecode text
2 Identify method in bytecode text

```
<com.connectsdk.service.NetcastTVService$1: void run()>
virtualinvoke $r13.<com.connectsdk.service.netcast.NetcastHttpServer: void start()>()
<com.connectsdk.service.netcast.NetcastHttpServer: void start()>
r0 := @this: com.connectsdk.service.netcast.NetcastHttpServer
```

```
Virtual methods   -
  #0              : (in Lcom/connectsdk/service/NetcastTVService$1;)
    name          : 'run'
    type          : '()V'
    access        : 0x0001 (PUBLIC)
    ......
    insns size    : 46 16-bit code units
13834c:                        |[13834c] com.connectsdk.service.NetcastTVService.1.run:()V
13835c: 5450 b417              |0000: iget-object v0, v5,
Lcom/connectsdk/service/NetcastTVService$1;.this$0:Lcom/connectsdk/service/NetcastTV
Service; // field@17b4
138360: 2201 d207              |0002: new-instance v1,
Lcom/connectsdk/service/netcast/NetcastHttpServer; // type@07d2
......
1383ac: 5400 1318              |0028: iget-object v0, v0,
Lcom/connectsdk/service/NetcastTVService;.httpServer:Lcom/connectsdk/service/netcast/N
etcastHttpServer; // field@1813
1383b0: 6e10 b930 0000         |002a: invoke-virtual {v0},
Lcom/connectsdk/service/netcast/NetcastHttpServer;.start:()V // method@30b9
1383b6: 0e00                   |002d: return-void
```

Fig. 3: Illustrating BackDroid's basic bytecode search process using a signature-based search example.

The basic signature-based search. As illustrated in Figure 3, we conduct the signature-based search in five steps, which are across not only the bytecode search space but also the program analysis space (with Soot). Given a callee method, we first translate its method signature from Soot's IR format to `dexdump`'s bytecode format to facilitate the search. With the transformed method signature, we then search the entire bytecode plaintext to locate all its invocation(s), as highlighted in the bottom of Figure 3. In the second step, we identify the corresponding method that contains the invocation found in the bytecode plaintext. Here it is `com.connectsdk.service.NetcastTVService.$1.run:()V`, where an inner class needs to add back the symbol "$". With this caller method signature (in bytecode format), we perform another format translation in the third step, and locate its Java method body via the program analysis in Soot. Next, we conduct a quick forward analysis via Soot to find the actual call site in the caller method body. With all these steps done, we finally connect a edge from the caller (site) to the callee method in SSG (self-contained slicing graph). Note that SSG is generated during the process of bytecode search and backward slicing. We will explain the details of SSG and its generation in §V-A.

After understanding this search process, we now turn to an important question not answered yet: which kinds of (callee) methods are suitable for the signature-based search. We call such methods *signature methods*. Typical signature methods include static methods[3] (either class or method is marked with the `static` keyword), private methods (similarly, methods declared with the `private` keyword), and constructors (e.g., `<init>` methods of a class). For some searches over child classes, we can also simply launch the signature-based search.

Searching over a child class. Suppose that the `NetcastHttpServer` class in Figure 3 has a child class called `Child Server`, we can still use the signature-based search but need to construct appropriate search signatures. Specifically, it depends on whether `ChildServer` overloads the callee method `void start()` or not. If it is not overloaded, an invocation of the callee method `start()` may also come from a child class object. Hence, besides the original signature search, we add one more signature search with the child class, namely `Lcom/connectsdk/service/netcast/ChildServer;.start:()V`. The returned caller(s) might come from both searches, or just one of them, depending on

[3] It is worth noting that although the static `<clinit>` method of a class is a signature method, it has to use a special search instead of the basic signature-based search, as we will explain in §IV-C.

how app developers invoke that particular callee method. On the other hand, if ChildServer does overload the start() method, we perform only one search with the original callee method signature because the child class search signature now corresponds to the overloaded child method only.

B. Advanced Search with Forward Object Taint Analysis

Although the basic search presented in the last subsection can handle many callee methods in an app bytecode, it is not effective for complex cases with superclasses, interfaces, callbacks, and asynchronous Java/Android flows. Note that these obstacles are different from the situation when they appeared in the previous research [10], [39], where the challenge was to determine object types instead of hindering the searches. We assume an artificial example where NetcastHttpServer.start() in Figure 3 has a superclass method called SuperServer.start() to explain the difficulty. Under this assumption, the original signature search may not reveal any valid callers, because developers may write code in this way: “*SuperServer server = new NetcastHttpServer(); server.start();*”. In this example, the bytecode signature of server.start() is Lcom/connectsdk/service/ netcast/SuperServer;.start:()V. Hence, searching with Netcast HttpServer’s method signature would hit nothing. But we also cannot use superclass SuperServer’s signature to launch the search, because it could return callers of the super method itself and other class methods that inherit from SuperServer. Similarly, if a callee method implements an interface, searching using the interface method signature would not work because an interface method might be implemented by arbitrary classes. Furthermore, searching over callbacks and asynchronous Java/Android flows could be even more difficult, because they employ different sub-method signatures for a pair of caller and callee methods.

We design a novel mechanism to accurately handle all these complex searches. The basic idea is that instead of directly searching for caller methods, we first search the callee class’s object constructor(s) that can be accurately located via the signature-based search. Right from those object constructors, we then perform forward object taint analysis until we detect the caller methods with the tainted object propagated into. We depict this process in Figure 4, using the same LG TV Plus app. This time the callee method is <com.connectsdk.service. NetcastTVService$1: void run()>, which continues the search flow in Fig. 3. We now present the involved four steps.

Searching for the object constructor. In step ⓿, we first determine that the target callee method requires advanced search, and then retrieve all its constructors. Here the callee class NetcastTVService$1 has only one constructor, i.e., <init>(com.connectsdk.service. NetcastTVService). We then launch a bytecode search using this signature to *accurately* locate that the constructor is initialized in a method called NetcastTVService: void connect(), as shown in step ❶.

Propagating object using taint analysis. In step ❷, we perform forward propagation of the located constructor object, i.e., $r11 in Figure 4, using taint analysis. Specifically, an object can be propagated via a definition statement, e.g., r0 := @parameter0: java.lang.Runnable, via an invoke statement, e.g., runInBackground($r11), or via a return statement. Therefore, we track only three kinds of statements, namely DefinitionStmt, InvokeStmt, and ReturnStmt.

Determining the ending method to stop. An important step is to determine at which *ending method* our forward analysis should stop. This is easy for the case of superclass, because we can simply stop at a tainted statement with the same sub-method signature as the callee method. However, it is difficult for the cases of interface, callback, and asynchronous flow, because their sub-method signature might be different from that in a callee method. Some previous works (e.g., [22], [43], [45]) used the pre-defined domain knowledge to connect those asynchronous flows, e.g., a common example is to connect Thread class’s start() and run() methods. However, in the example of Figure 4, it will miss the ending method Executor.execute(). We also tried the flow mapping provided by EdgeMiner [14] but found that it could contain over-estimated asynchronous flows, such as connecting 2,446 caller methods (e.g., WaveView.<init>() and Thread.<init>()) to only a single callee method MediaPlayer$OnErrorListener.onError().

To better determine the ending method, we propose a mechanism that does not rely on prior knowledge but leverages interface’s class type as an indicator to find a tainted parameter that is directly propagated from the original constructor object. For example, in Figure 4, since the interface class type is java.lang.Runnable, we determine which on-path Java/Android API call contains a tainted parameter that belongs to this class type. Our forward analysis thus propagates the original $r11 object along the path highlighted, and eventually reaches at r0 in the statement Executor.execute(r0). Hence, it is the ending method we need to identify in this example.

Maintaining and returning a call chain. Different from the basic search that returns just one call site, here we need to maintain and return a call chain, i.e., the chain from NetcastTVService.connect() to Util.runInBackground (Runnable) and further to Util.runInBackground(Runnable,boolean). Assuming that we just return one call site and one caller method, the further backward search of Util. runInBackground(Runnable,boolean) may return multiple search results or flows, because this search is now independent. However, the fact is that only the flow shown in Fig. 4 could eventually trace back to the constructor object. Therefore, to avoid mis-added flows, we maintain a call chain during the forward taint analysis.

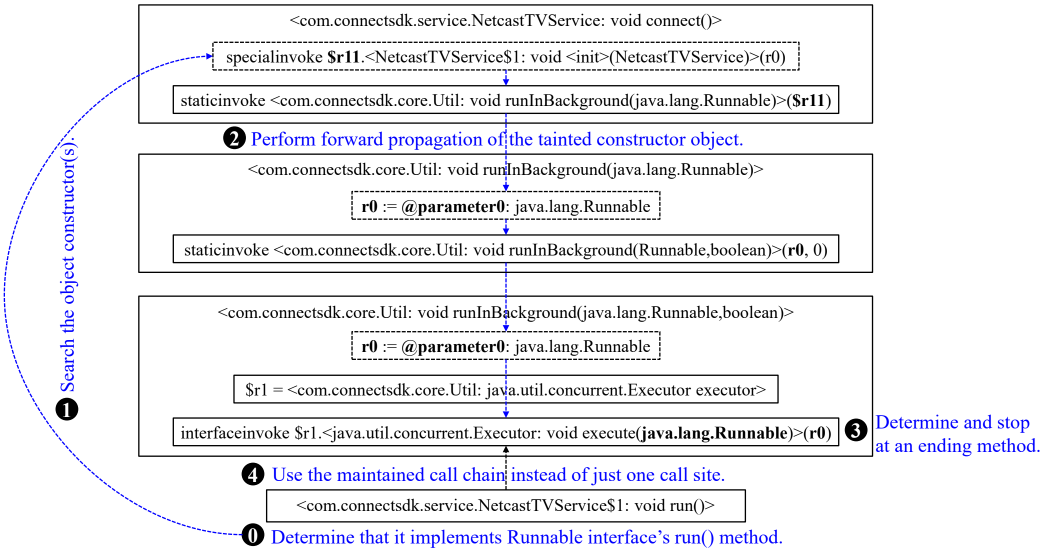

<com.connectsdk.service.NetcastTVService: void connect()>
specialinvoke $r11.<NetcastTVService$1: void <init>(NetcastTVService)>(r0)
staticinvoke <com.connectsdk.core.Util: void runInBackground(java.lang.Runnable)>($r11)
❷ Perform forward propagation of the tainted constructor object.
<com.connectsdk.core.Util: void runInBackground(java.lang.Runnable)>
r0 := @parameter0: java.lang.Runnable
staticinvoke <com.connectsdk.core.Util: void runInBackground(Runnable,boolean)>(r0, 0)
<com.connectsdk.core.Util: void runInBackground(java.lang.Runnable,boolean)>
r0 := @parameter0: java.lang.Runnable
$r1 = <com.connectsdk.core.Util: java.util.concurrent.Executor executor>
interfaceinvoke $r1.<java.util.concurrent.Executor: void execute(java.lang.Runnable)>(r0)
❸ Determine and stop at an ending method.
❹ Use the maintained call chain instead of just one call site.
<com.connectsdk.service.NetcastTVService$1: void run()>
⓿ Determine that it implements Runnable interface's run() method.
❶ Search the object constructor(s).

Fig. 4: Using advanced search with forward object taint analysis to uncover a caller chain of an interface method, `NetcastTVService$1.run()`. Note that step 1 uses the basic signature search in §IV-A, the process of which is thus skipped.

C. Special Search over Static Initializers

The basic and advanced searches presented in the last two subsections are useful in most scenarios, but they are not for handling some special search challenges. We thus further propose several special search mechanisms. In this subsection, we present a recursive search for static initializers.

A static initializer is the static `<clinit>` method of a class, which may occasionally appear in a call path BackDroid is backtracking. Resolving its caller methods is a must for determining whether the corresponding call path is reachable from entry points or not. However, it is impossible to directly search static initializers' callers, because they are never (explicitly) invoked by any app bytecode. Instead, `<clinit>` methods are only *implicitly* executed by Java/Android virtual machine (VM) when the corresponding classes are loaded to the VM. Hence, we design a special search mechanism to handle these *on-path* static initializers. Additionally, in §V-A, we will also show how to add back *off-path* static initializers into our slicing graph for the complete dataflow analysis.

To illustrate our approach, we use another concrete example, the popular Heyzap advertisement library (now acquired by Fiber) embedded in many apps. In this example, when BackDroid backtracks the `setHostnameVerifier()` sink API invoked by `com.heyzap.http.MySSLSocketFactory`, the analysis comes to a static initializer of the `com.heyzap.internal.APIClient` class. However, a further search of the `APIClient.<clinit>()` initializer method would hit nothing, as we have explained earlier. Normally, a forward whole-app analysis approach can track such initializer methods whenever it encounters a field or a method of the `APIClient` class, but it is unrealistic here to backward search all fields and methods of `APIClient`. To address this unique issue in BackDroid, we propose a different strategy that performs recursive searches to determine only the control-flow reachability of a targeted `<clinit>` method. This is reasonable because `<clinit>` have no dataflow propagation due to no parameter passing.

Formally, for a static initializer `SI.<clinit>()`, our recursive search works as follows. BackDroid first launches a search to find out a set of classes $C = \{c_1, \cdots, c_n\}$ that invoke the `SI` class. It then determines whether any class c_i in this set is an entry component registered in the app manifest. In the case of Heyzap's `APIClient` class, since no c_i is an entry class, BackDroid continues to perform the class search of each c_i to determine whether any of its contained classes is an entry class. This process repeats until no more new class is searched out or entry class is identified. For example, the `APIClient` class is invoked by the class `com.heyzap.house.model.AdModel`, which is further used by an entry class called `com.heyzap.sdk.ads.HeyzapInterstitialActivity`. Therefore, we consider the initializer `APIClient.<clinit>()` reachable from entry points and the associated call path also valid.

Through the evaluation in §VI, we have obtained some convincing data in real apps to support our design in this subsection. Among 37 unique static initializers that are identified by our recursive search as reachable, we find that all of them are reachable from entry components.

D. Special Search over Android ICC

In this subsection, we present another special search mechanism to track data flows over Android-specific inter-component communication (ICC), a fundamental code/data collaboration mechanism on Android [32], [42].

Our search is based on the inner working mechanism of Android ICC. Specifically, ICC is different from typical API calls because it relies on its `Intent` parameter values to dynamically determine a target callee. A callee could be *explicitly* specified by setting the target component class (e.g., via `Intent i = new Intent(activity, HttpServerService.class);`), or *implicitly* specified by setting an `Intent` action that will be delivered by the operating system (OS) to the target component.

Based on this observation, we propose a two-time search mechanism to handle ICC. The basic idea is to launch two searches: one for searching ICC calls (e.g., `startService()`), and the other for searching ICC parameters. For the explicit ICC, the second parameter search directly searches component class names, e.g., `const-class .*, Lcom/lge/app1/fota/HttpServerService;`, while for the implicit ICC, we search `Intent` action names instead. After performing them, we merge the two search results and check whether an ICC call satisfies both. If such an ICC call exists, it is the caller method we are looking for.

E. Special Search over Android Lifecycle Handlers

The last specific search challenge is how to search over Android lifecycle handlers, e.g., the `onStart()` and `onResume()` methods in `Activity` components. Each lifecycle handler could be an entry function, and they can be executed in multiple orders [10]. Our strategy is to first determine whether the dataflow tracking finishes when reaching at a lifecycle handler. If it does, we have no need to launch further search, since the tracked lifecycle handler is already an entry method. Otherwise, we conduct a special search that leverages existing domain knowledge [10] to further track other lifecycle handlers that invoke the callee handler. Since there are only four kinds of Android components, we can simply use domain knowledge to handle all lifecycle handlers.

V. Adjusting Traditional Backward Slicing and Forward Analysis

With the new paradigm of inter-procedural analysis enabled by our on-the-fly bytecode search, we further adjust the traditional backward slicing and forward analysis to provide the dataflow tracking capability [10], [39] in our context. As mentioned earlier in §III, we generate a new slicing structure, self-contained slicing graph (SSG), during the backtracking, and perform forward constant and points-to propagation over SSG to calculate valid dataflow facts.

A. Generating a Self-contained Slicing Graph (SSG)

Since our bytecode search reveals only inter-procedural relationships and we do not have a whole-app graph, we need our own graph structure to record all the slicing and inter-procedural information during the backtracking. This graph essentially reflects the partial app paths visited by our on-the-fly analysis, and the information stored in it can be used by forward analysis to recover the complete dataflow representation of target sink API calls. Below we first describe this graph structure, and then present the implementation challenges we overcame to generate it.

Defining a self-contained graph structure to record all the slicing and inter-procedural information. We propose a self-contained graph structure called *self-contained slicing graph* (SSG) to cover all the slicing and inter-procedural information generated during our backtracking. With this structure, we aim to cover the slicing information across different parameters tracked, different paths traced, and different kinds of bytecode instructions, besides recording the inter-procedural relationships uncovered by our bytecode search. Hence, it is different from the individual path-like slices generated in typical Android slicing tools (e.g., [12], [24], [49]). We currently design each SSG corresponding to one unique sink API call, and we will also provide the per-app SSG in the future. Figure 5 shows an example SSG (simplified for readability) that is automatically generated by BackDroid for the app `com.studiosol.palcomp3`.

Backward taint analysis over fields and arrays for the SSG generation. With the SSG structure defined, we then perform *backward* taint analysis to generate SSGs. Compared to the forward taint analysis in Amandroid and FlowDroid, backward taint analysis is more difficult because it reverses the normal program execution and thus has no insights on the earlier execution of tainted variables. This problem is particularly noticeable for fields and arrays, and we handle them as follows. Specifically, for an instance field to be tainted, e.g., `r0.<com.studiosol.util.NanoHTTPD: int myPort>` in Figure 5, we add not only the instance field itself to the taint set but also its class object (i.e., `r0`) so that we can trace the same field no matter the class object gets aliased or across method boundaries. Moreover, when an instance field needs to be untainted, we first remove `obj.field` from the taint set and further detect whether there are more fields for the same instance. If there are no other such fields, we remove `obj` from the taint set as well. Arrays and `Intent` objects are handled in a similar way. Hence, we skip the details here.

Adding off-path static initializers into SSG on demand. In §IV-C, we introduced how to search over *on-path* static initializers. Here we continue to explain how to accordingly add *off-path* static initializers, i.e., those not in the backtracking paths. Specifically, after the main taint process is done, if there are still unresolved static fields in the SSG's taint map, we retrieve the corresponding classes and obtain their `<clinit>` methods that are only implicitly executed by the Java/Android virtual machine. We then perform the backward taint analysis of these `<clinit>` methods, and add only relevant statements into a special track of SSG. Figure 5 presents such a track for the `MP3LocalServer.<clinit>()` method, which captures the value of an unresolved static field, `<com.studiosol.palcomp3.MP3LocalServer: int PORT>`. During the forward analysis, we first analyze this special static track and then handle the main track of SSG.

B. Forward Constant and Points-to Propagation over SSG

After producing a complete SSG, our forward analysis iterates through each SSG node, analyzes its semantic, and

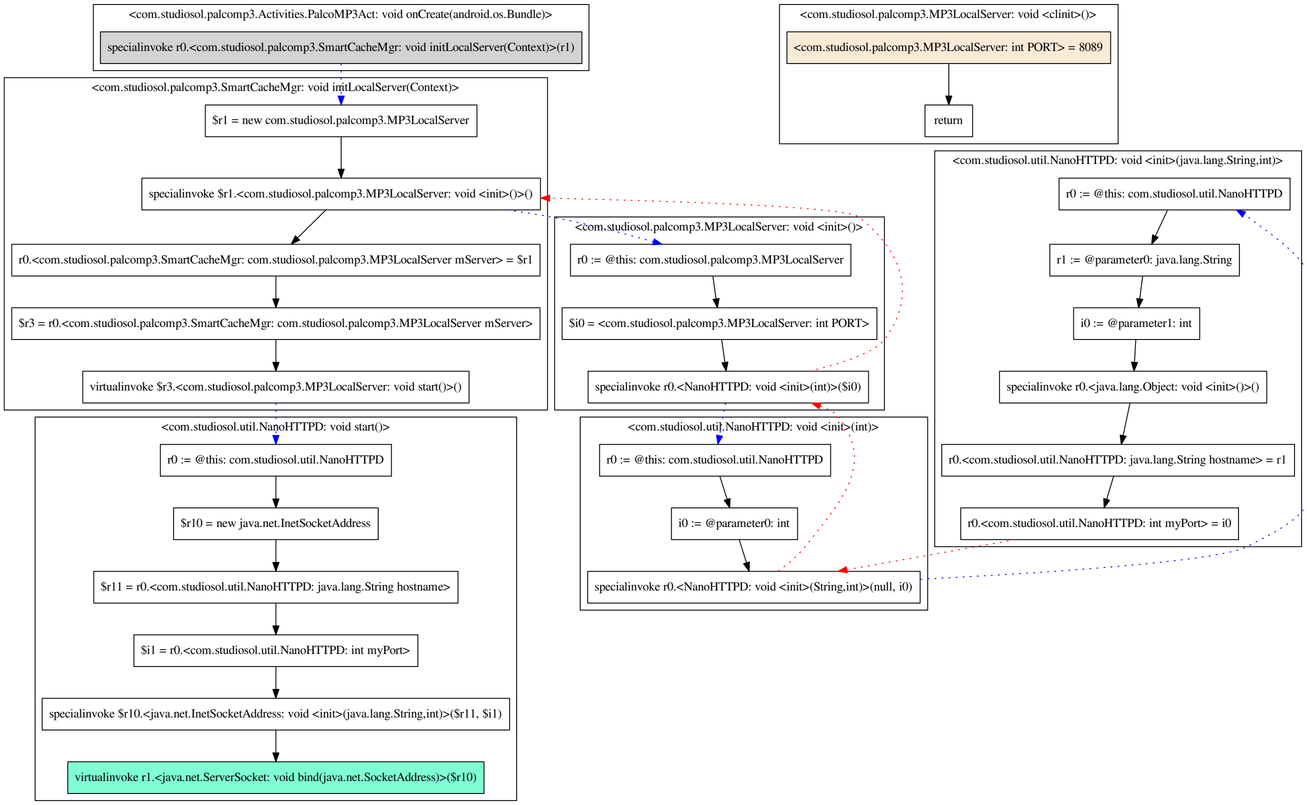

<com.studiosol.palcomp3.Activities.PalcoMP3Act: void onCreate(android.os.Bundle)>
specialinvoke r0.<com.studiosol.palcomp3.SmartCacheMgr: void initLocalServer(Context)>(r1)
<com.studiosol.palcomp3.SmartCacheMgr: void initLocalServer(Context)>
$r1 = new com.studiosol.palcomp3.MP3LocalServer
specialinvoke $r1.<com.studiosol.palcomp3.MP3LocalServer: void <init>()>()
r0.<com.studiosol.palcomp3.SmartCacheMgr: com.studiosol.palcomp3.MP3LocalServer mServer> = $r1
$r3 = r0.<com.studiosol.palcomp3.SmartCacheMgr: com.studiosol.palcomp3.MP3LocalServer mServer>
virtualinvoke $r3.<com.studiosol.palcomp3.MP3LocalServer: void start()>()
<com.studiosol.palcomp3.MP3LocalServer: void <clinit>()>
<com.studiosol.palcomp3.MP3LocalServer: int PORT> = 8089
return
<com.studiosol.palcomp3.MP3LocalServer: void <init>()>
r0 := @this: com.studiosol.palcomp3.MP3LocalServer
$i0 = <com.studiosol.palcomp3.MP3LocalServer: int PORT>
specialinvoke r0.<NanoHTTPD: void <init>(int)>($i0)
<com.studiosol.util.NanoHTTPD: void <init>(java.lang.String,int)>
r0 := @this: com.studiosol.util.NanoHTTPD
r1 := @parameter0: java.lang.String
i0 := @parameter1: int
specialinvoke r0.<java.lang.Object: void <init>()>()
r0.<com.studiosol.util.NanoHTTPD: java.lang.String hostname> = r1
r0.<com.studiosol.util.NanoHTTPD: int myPort> = i0
<com.studiosol.util.NanoHTTPD: void start()>
r0 := @this: com.studiosol.util.NanoHTTPD
$r10 = new java.net.InetSocketAddress
$r11 = r0.<com.studiosol.util.NanoHTTPD: java.lang.String hostname>
$i1 = r0.<com.studiosol.util.NanoHTTPD: int myPort>
specialinvoke $r10.<java.net.InetSocketAddress: void <init>(java.lang.String,int)>($r11, $i1)
virtualinvoke r1.<java.net.ServerSocket: void bind(java.net.SocketAddress)>($r10)
<com.studiosol.util.NanoHTTPD: void <init>(int)>
r0 := @this: com.studiosol.util.NanoHTTPD
i0 := @parameter0: int
specialinvoke r0.<NanoHTTPD: void <init>(String,int)>(null, i0)

Fig. 5: An SSG automatically generated by BackDroid, where the green block is a sink API call and the gray block is an entry.

propagates dataflow facts through the constant [38] and points-to [26] propagation during the graph traversal. We now explain these forward analysis steps over our new SSG structure.

Overall traversal process over SSG. As mentioned in §V-A, an SSG includes two tracks, the special static field track and the normal track. Our traversal always starts with the static field track so that we can resolve fields referred in the normal track. For each track, we first retrieve a set of tail nodes (e.g., the starting blocks in Figure 5) and initialize analysis from each of them. To record dataflow facts generated by our analysis, we maintain fact maps for each analysis flow, but we use only one global fact map for all static fields.

Whenever we reach at a new SSG node, we perform graph traversal as follows. First, we determine whether the node is an initial SSG node with a sink API call (e.g., the ending block in Figure 5); and if it is, we correlate and output dataflow facts of all tainted parameters. For a normal SSG node, we first jump into its invoked methods if any. After that, we analyze the node itself and move to the next node(s).

Propagating constant and points-to information. To facilitate the dataflow propagation, we maintain a fact map to correlate each variable with its dataflow fact. Propagating constant facts among different variables is easy — just retrieve the value from an old variable and assign it to a new variable in the fact map. To propagate object points-to information, we design an object structure called `NewObj` to preserve the original points-to information along flow paths. Each `NewObj` object contains a pointer to its constructor class, a map of member objects (in any class type) and their reference names. Then we just need to propagate `NewObj` objects along flow paths so that all corresponding objects being traced can point to the same `NewObj` object. Inner members of `NewObj` can also be updated by checking classes' `<init>` methods or any other value-assignment statements. Besides the class objects' points-to information, we define an `ArrayObj` object to wrap the points-to information of array expression and its array map between indexes and values.

VI. Evaluation

In this section, we evaluate the efficiency and efficacy of BackDroid in analyzing modern apps. In particular, we compare BackDroid with Amandroid [39], [40], the state-of-the-art Android static dataflow analysis tool. Both Amandroid and BackDroid support the dataflow analysis of all kinds of sink-based analysis problems, such as API misuse (e.g., [13], [15]) and malware detection (e.g., [23], [30]). In contrast, FlowDroid focuses only on the privacy leak detection that involves both source and sink APIs, and thus it is not compared here. Nevertheless, by comparing our BackDroid's result presented in this section with the call graph generation result of FlowDroid in §II-C, we find that BackDroid still performs much faster (2.13min v.s. 9.76min, on average).

A. Experimental Setup

To fairly evaluate both BackDroid and Amandroid, we select two common and serious sink-based problems, crypto and SSL/TLS misconfigurations, which were also recently tested by Amandroid in [40]. In both cases, the root cause is due to insecure parameters. For example, the ECB mode is used to create the `javax.crypto.Cipher` instance [15], [17], [29] and the insecure parameter `ALLOW_ALL_HOSTNAME_VERIFIER` is used in `setHostnameVerifier()` [18], [19], [37]. Note that these two kinds of sink APIs are frequently invoked in our tested apps — on average, 21 sink API calls in each app of our dataset. In this way, we can stress-test the performance of BackDroid even though we are targeting at just two problems. In the rest of this subsection, we describe the dataset tested, computing environment used, and tool parameters configured.

Dataset. We use a set of modern popular apps that satisfy two conditions: they (i) have at least one million installs each, and (ii) were updated in recent years. Specifically, we first select all such 3,178 apps in our app repository (see Table I in §II) as our basic dataset. However, not all of them contain the specific sink APIs, so we pre-search them to filter out the apps with all three target sink APIs. This is to help Amandroid avoid unnecessary analysis since it has no bytecode search capability. Eventually, we use the searched 144 apps for experiments.

Environment. For the computing environment, we use a desktop PC with Intel i7-4790 CPU (3.6GHZ, eight cores) and 16GB of physical memory. Note that a memory configuration with 16GB or less is often used in many previous studies, e.g., [34], [36], [44], [46]. To guarantee sufficient memory for the OS itself, we assign 12GB RAM to the Java VM heap space in running Amandroid. To demonstrate that BackDroid is not sensitive to the amount of available memory, we use only 4GB (i.e., `-Xmx4g`). The OS is 64-bit Ubuntu 16.04, and we use Java 1.8 and Python 2.7 to run the experiments.

Tool configuration. Both Amandroid and FlowDroid need to configure a set of parameters to balance their performance and precision. In contrast, BackDroid does not require specific parameter configuration, since we have programmed its maximum capability in the code. In this paper, we use the default Amandroid parameters (as in its `config.ini` file), and use the latest Amandroid 2.0.5 that supports the inter-procedural API misuse analysis[4]. We also give it sufficient running time with a large timeout of 300 minutes per app.

B. Performance Results

Out of the 144 apps analyzed, both Amandroid and BackDroid successfully finished the analysis of 141 apps. For Amandroid, the three failures are all due to its errors in parsing the manifest, while for BackDroid, two failures are caused by the format transformation from bytecode to IR (in `com.kbstar.liivbank` and `com.lguplus.paynow`), and one failure is a Soot bug in processing the `com.lcacApp` app. In our evaluation, we thus do not count these failed apps.

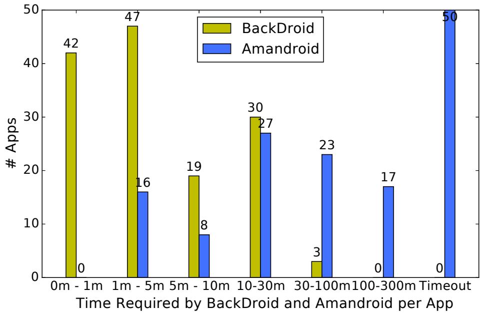


Fig. 6: The distribution of time in BackDroid and Amandroid.

Figure 6 shows the distribution of analysis time used by BackDroid and Amandroid, respectively. By correlating them, we make the following three observations:

First, *BackDroid's has no any timed-out failure, while the timed-out failure rate in Amandroid is as high as 35%*. As shown in Figure 6, 50 apps still timed out in Amandroid, even though we have set a considerably large timeout for Amandroid (five hours for each single app). This 35% (50/141) timed-out failure rate indicates that Amandroid is out of control in analyzing modern apps. In contrast, only three apps in BackDroid exceeded 30 minutes, with 35min, 39min, and 81min, respectively. This suggests that BackDroid's analysis time is under control, even when we are analyzing large apps.

Second, *BackDroid can quickly finish the analysis of most of the apps, with 77% of apps analyzed within 10 minutes and even with one-third of apps finished within just one minute*. After analyzing the cases of long analysis times, we now turn to the apps with short ones. BackDroid requires just one minute to analyze 30% (42) of apps each, and as high as 77% (108) of apps are quickly finished within 10 minutes each. This gives BackDroid great potential to be deployed by app markets for online vetting. In contrast, only 17% (24) of apps can be analyzed by Amandroid within the same time slot, and no app could be finished within one minute.

Third, *the overall performance of BackDroid, in terms of the median time, is 37 times faster than that in Amandroid*. We further compare the overall performance between BackDroid and Amandroid. Since there are 50 timed-out failures in Amandroid's result, measuring the average is not reliable. We thus analyze the median time, and find that the performance gap between the two tools is quite significant: the overall median time of all 141 apps analyzed in BackDroid is 37 times faster than that in Amandroid (2.13min versus 78.15min).

Besides the three comparisons above, in Figure 7, we further plot the relationship between the number of sink API calls analyzed in each app and BackDroid's analysis time. We can see that the majority are at a speed faster than 30 seconds per sink call. For example, we can draw a straight line from the coordinate (0, 0) to the coordinate (60, 1800), and only around ten dots are above this line. Under this linear trend, BackDroid is expected to finish the analysis of 100 sink calls with around 50 minutes. Indeed, all apps, except one, were analyzed within 40 minutes. The only outlier, Huawei Health, costed 81min for its 121 sink API calls, which is still much faster than the 300-minute timeout in Amandroid.

[4] Amandroid after version 2.0.5 uses only the *intra*-procedural analysis to analyze API misuse; see https://github.com/arguslab/Argus-SAF/issues/55.

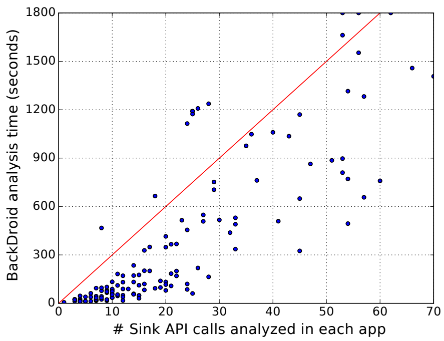


Fig. 7: The relationship between the number of sink API calls in each app and BackDroid's analysis time.

C. Detection Results

After comparing BackDroid's and Amandroid's efficiency, we now analyze and compare their detection accuracy.

Vulnerabilities detected by Amandroid but not BackDroid. We first analyze whether BackDroid could achieve a close detection rate for the app vulnerabilities detected by Amandroid. For the crypto API usage, Amandroid detected seven apps with insecure ECB mode. BackDroid can accurately detect all of them. The buggy apps include the popular Adobe Fill & Sign app (`com.adobe.fas`) and a bank app called IDBI Bank GO Mobile+ (`com.snapwork.IDBI`). Both apps must guarantee a secure encryption in their design. We have reported these risky issues to their vendors (but their willingness to fix these issues is low). We also find that to detect these seven vulnerable apps, Amandroid spent a total of 8.53 hours (73min on average), whereas BackDroid required only 8.78 minutes (1.26min each), around 60 times faster.

Compared to the crypto API misuse, Amandroid detected more SSL misconfigurations in our dataset, with 23 apps discovered with the wrong SSL hostname verification. However, our diagnosis shows that six of them are false positives. Among the 17 true positives, BackDroid detected 15 of them and failed on two apps. The detailed diagnosis results are:

- *Four false positives from the ArmSeedCheck library*: Amandroid reported four buggy apps with a library called `com.skt.arm.ArmSeedCheck`. However, three of them do not trace back to entry functions, and the sink flow of one app (`kemco.hitpoint.machine`) originates from an `Activity` component (`jp.kemco.activation.TstoreActivation`) not in manifest.
- *Two false positives from the qihoopay library*: Two vulnerable apps with the `com.qihoopay.insdk.utils.HttpUtils` library were reported by Amandroid. However, their sink flows similarly come from a unregistered and thus deactivated `Activity` component.
- *Two false negatives in BackDroid*: Unfortunately, BackDroid failed on the `com.gta.nslm2` and `com.wb.goog.mkx` apps. The root cause for both cases is that they do not directly invoke the system sink APIs, which makes our initial bytecode search step fail to locate their sink API calls. We can address this issue by checking the class hierarchy also in the initial search.

Vulnerabilities detected by BackDroid but not Amandroid. We further find that for some apps, BackDroid can achieve better detection than Amandroid. In particular, BackDroid discovered 54 additional apps with potentially insecure ECB and SSL issues that were not detected by Amandroid. By analyzing them, we identify four important factors (the last two, with 18 apps, are fully due to Amandroid's inaccuracy):

- *Timed-out failures*: 28 of the 54 failed apps were due to the timeouts, where Amandroid did not finish their analysis even after running 300 minutes each.
- *Skipped libraries*: Since Amandroid by default skips the analysis of some popular libraries, it failed to detect the ECB/SSL issues in eight apps, which use the skipped Java packages from Amazon, Tencent, and Facebook.
- *Unrobust handling of implicit flows*: We surprisingly find Amandroid not robust as BackDroid in handling certain asynchronous flows and callbacks. For example, Amandroid failed to connect the flow from `AsyncTask.execute()` to `doInBackground()` and the callback from `setOnClickListener()` to `onClick()`.
- *Occasional errors in whole-app analysis*: By inspecting Amandroid's debug logs, we observed some occasional errors (e.g., "cannot find procedure" and "key not found") during the analysis of Amandroid, which cause 10 apps failed. By nature, it is more error-prone for Amandroid's whole-app analysis than BackDroid's targeted analysis.

D. Threats To Validity

While we extensively tested the performance and accuracy in §VI-B and §VI-C, the apps evaluated were collected in late 2018 (i.e., the same set of apps used by FlowDroid in §II-C). Using more recent apps for evaluation will be our future work. Moreover, as similar to typical static analysis tools, BackDroid currently cannot track data across implicit (control) flows.

VII. Conclusion

In this paper, we proposed a new paradigm of targeted interprocedural analysis by combining traditional program analysis with our on-the-fly bytecode search. We implemented this technique into a tool called BackDroid for targeted and efficient security vetting of modern Android apps. We overcame unique challenges to search over Java polymorphism, asynchronous flows, callbacks, static initializers, and Android intercomponent communication. We also adjusted the traditional backward slicing and forward analysis over a structure called self-contained slicing graph (SSG) for the complete dataflow tracking. Our experimental results showed that BackDroid is 37 times faster than Amandroid and has no timeout (v.s. 35% in Amandroid) while maintaining close or even better detection effectiveness. In the future, we will extend BackDroid to other non-sink-based problems, such as privacy leak detection.

Acknowledgement. We would like to thank our shepherd, Katinka Wolter, and all the reviewers for their valuable comments and suggestions. This research/project is partially supported by the Singapore National Research Foundation under the National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001).

References

[1] "Disassemble Android dex files," http://blog.vogella.com/2011/02/14/disassemble-android-dex/, 2011.

[2] "[soot-list] FlowDroid: call graph doesn't look context sensitive," https://mailman.cs.mcgill.ca/pipermail/soot-list/2016-March/008410.html, 2016.

[3] "Using Geometric Encoding based Context Sensitive Points to Analysis (geomPTA)," https://tinyurl.com/geomPTA, 2016.

[4] "FlowDroid releases on GitHub," https://github.com/secure-software-engineering/FlowDroid/releases, 2019.

[5] "Configure Apps with Over 64K Methods," https://developer.android.com/studio/build/multidex.html, 2020.

[6] "T.J. Watson Libraries for Analysis (WALA)," https://github.com/wala/WALA, 2020.

[7] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. ACM MSR*, 2016.

[8] S. Arzt, "Static data flow analysis for Android applications," *PhD Dissertation*, December 2016.

[9] S. Arzt and E. Bodden, "StubDroid: Automatic inference of precise data-flow summaries for the Android framework," in *Proc. ACM ICSE*, 2016.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM PLDI*, 2014.

[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. ACM ICSE*, 2015.

[12] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-Droid: Leveraging Android app analysis with static slice optimization," in *Proc. ACM AsiaCCS*, 2016.

[13] A. Bianchi, Y. Fratantonio, A. Machiry, C. Krugel, G. Vigna, S. P. H. Chung, and W. Lee, "Broken Fingers: On the usage of the fingerprint API in Android," in *Proc. ISOC NDSS*, 2018.

[14] 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 *Proc. ISOC NDSS*, 2015.

[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proc. ACM CCS*, 2013.

[16] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proc. ISOC NDSS*, 2011.

[17] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. USENIX Security*, 2011.

[18] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love Android: An analysis of Android SSL (in)security," in *Proc. ACM CCS*, 2012.

[19] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *Proc. ACM CCS*, 2012.

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. Springer TRUST*, 2012.

[21] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. ISOC NDSS*, 2015.

[22] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. ISOC NDSS*, 2012.

[23] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. ACM MobiSys*, 2012.

[24] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing Droids: Program slicing for Smali code," in *Proc. ACM SAC (Symposium on Applied Computing)*, 2013.

[25] P. Lam, E. Bodden, O. Lhotk, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. ACM ICSE*, 2015.

[26] O. Lhotak and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proc. Springer Compiler Construction*, 2003.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM CCS*, 2012.

[29] S. Ma, D. Lo, T. Li, and R. H. Deng, "CDRep: Automatic repair of cryptographic misuses in Android applications," in *Proc. ACM AsiaCCS*, 2016.

[30] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building markov chains of behavioral models," in *Proc. ISOC NDSS*, 2017.

[31] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: an empirical study of frequently-updated mobile apps in the Google Play store," *Springer Empirical Software Engineering*, vol. Volume 21, Issue 3, 2015.

[32] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. Usenix Security*, 2013.

[33] X. Pan, Y. Cao, X. Du, B. He, G. Fang, and Y. Chen, "FlowCog: Context-aware semantics extraction and analysis of information flow leaks in Android apps," in *Proc. USENIX Security*, 2018.

[34] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps," in *Proc. ISOC NDSS*, 2017.

[35] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects," in *Proc. ACM CCS*, 2019.

[36] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The misuse of Android Unix domain sockets and security implications," in *Proc. ACM CCS*, 2016.

[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 *Proc. ISOC NDSS*, 2014.

[38] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. Volume 13, Issue 2, 1991.

[39] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM CCS*, 2014.

[40] ——, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. Volume 21, Issue 3, 2018.

[41] M. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. ISOC NDSS*, 2016.

[42] D. Wu, Y. Cheng, D. Gao, Y. Li, and R. H. Deng, "SCLib: A practical and lightweight defense against component hijacking in Android applications," in *Proc. ACM CODASPY*, 2018.

[43] D. Wu, X. Luo, and R. K. C. Chang, "A sink-driven approach to detecting exposed component vulnerabilities in Android apps," *CoRR*, vol. abs/1405.6282, 2014.

[44] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. ACM ICSE*, 2015.

[45] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM CCS*, 2014.

[46] M. Zhang and H. Yin, "AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications," in *Proc. ISOC NDSS*, 2014.

[47] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *Proc. IEEE Symposium on Security and Privacy*, 2020.

[48] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symposium on Security and Privacy*, 2012.

[49] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in Android apps," in *Proc. ACM WiSec*, 2015.

[50] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proc. IEEE Symposium on Security and Privacy*, 2019.