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

Daoyuan Wu[1], Debin Gao[2], Robert H. Deng[2] , and Rocky K. C. Chang[3]

https://github.com/VPRLab/BackDroid

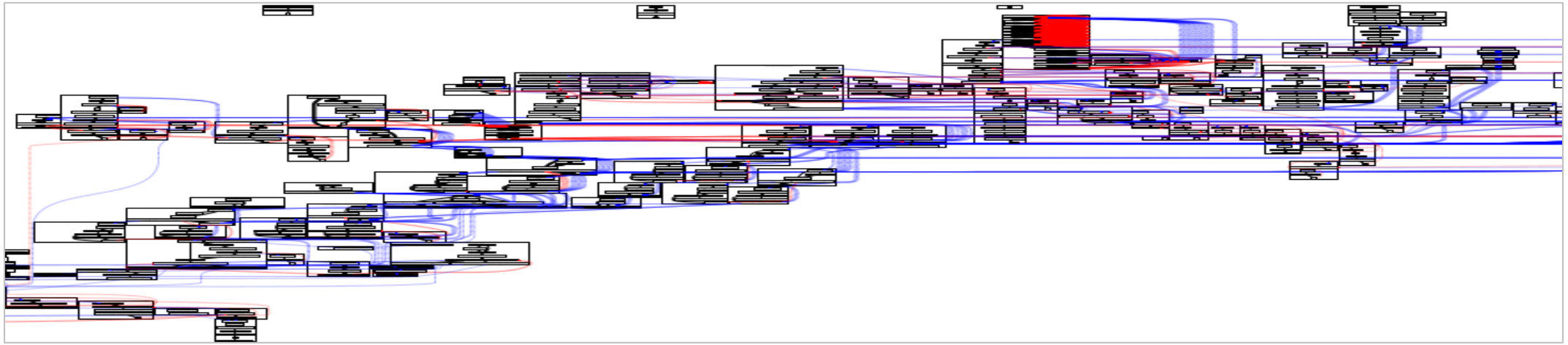
1 香港中文大學 The Chinese University of Hong Kong

2 SMU SINGAPORE MANAGEMENT UNIVERSITY

3 THE HONG KONG POLYTECHNIC UNIVERSITY 香港理工大學

State-of-the-art Android Static Tools

- AmanDroid [CCS'14, 413 cites] and FlowDroid [PLDI'14, 1,867 cites]
  - Both perform the **whole-app** **inter-procedural analysis** that starts from all entry points and ends in all reachable code nodes.



Comprehensive: all forward analysis could be built upon

Ignore the need of **targeted** analysis, and often **Expensive**

Previous Tests on Amandroid and FlowDroid

- For relatively **small** apps:

Apps under 5MB
in AppContext [ICSE'15]

- 16.1% of 1,002 apps exceeded the 80-minute timeout;
- **11min each** for the rest of apps.

Apps with an average size of 8.4MB
in HSOMiner [NDSS'17]

- 8.4% of 3K apps exceeded the 60-minute timeout;
- **13min each** for the rest of apps.

- Even with 730 GB of RAM and 64 CPU cores:
  - "*the server sometimes used all its memory, running on all cores for more than 24 hours to analyze one single Android app*" [ICSE'15].
- Industrial reports:
  - "This code runs for more than 5 hours to analyze an apk that is only 12.4M" #14
  - "Although I kept the analysis running for 72 hrs (with 28 GB memory), it seems like it's stuck being unable to find any entry points" FlowDroid Issue #310

The Upscaling Trend of Modern App Sizes

A summary of average and median app sizes over a period of five years.

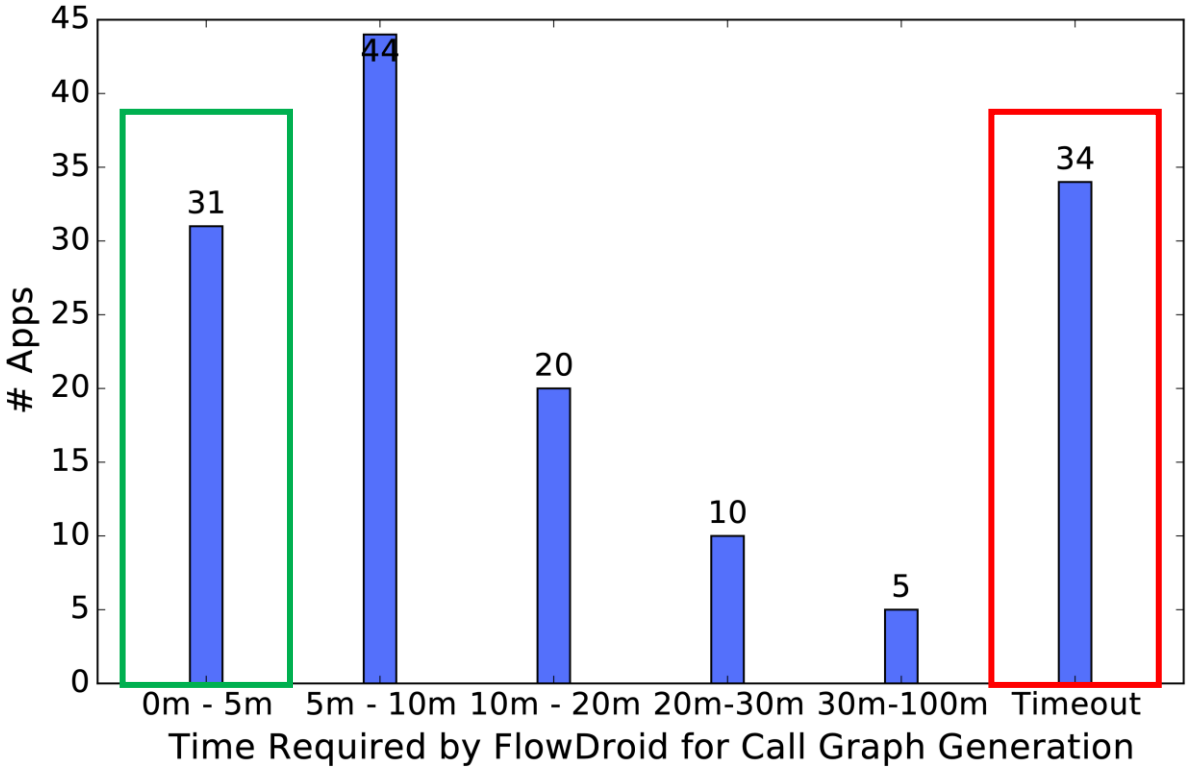
Year	Average Size	Median Size	# Samples
2014	13.8MB	8.4MB	2,840
2015	18.8MB	12.4MB	1,375
2016	21.6MB	16.2MB	3,510
2017	32.9MB	30.0MB	1,706
2018	42.6MB	38.0MB	3,178

X 3 **X 4.5**

Generating Whole-app Call Graphs for Modern Apps

- With modern apps, we re-evaluate **the cost** of generating a whole-app call graph using FlowDroid 2.7.1 (without the subsequent dataflow analysis):
  - 144 modern apps with the average size of 41.5MB, under the **same hardware configuration** as for our experiments of Amandroid and BackDroid later.

The median time of call graph generation in FlowDroid is **still around 10 minutes** (9.76min) per app



24% of the apps reached the timeout of 5 hours, causing **no result** for those 34 modern apps

Our Work

- Explore **a new paradigm of targeted** (vs. the traditional whole-app) inter-procedural analysis that can
  - skip irrelevant code and focus only on the flows of security-sensitive sink APIs.

- Propose a new technique called **on-the-fly bytecode search**,
  - which searches the disassembled app bytecode plaintext **just in time** when a caller method 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 modern Android apps.

An Overview of BackDroid's Analysis Process

Original Bytecode

dexdump

Dumped Bytecode Plaintext

Source for search

Bytecode Search

Entry function

<com.free.vpn.unblock.sites.proxybrowser.activity.SplashScreen: void onResume()>

virtualinvoke $r11.<com.free.vpn.unblock.sites.proxybrowser.activity.ax: android.os.AsyncTask execute(java.lang.Object[])>($r12)

virtualinvoke r0.<com.free.v

<com.free.vpn.unblock.sites.proxybrowser.activity.ax: void onPostExecute(java.lang.Object)>

virtualinvoke $r3.<com.free.vpn.unblock.sites.proxybrowser.activity.SplashScreen: void c()>()

<com.free.vpn.unblock.sites.proxybrowser.activity.SplashScreen: void c()>

virtualinvoke $r1.<com.free.vpn.unblock.sites.proxybrowser.activity.aw: android.os.AsyncTask execute(java.lang.Object[])>($r2)

<com.free.vpn.unblock.sites.proxybrowser.activity.aw: void onPostExecute(java.lang.Object)>

virtualinvoke $r4.<com.free.vpn.unblock.sites.proxybrowser.m.o: void run()>()

<com.free.vpn.unblock.sites.proxybrowser.m.o: void run()>

$r2 = staticinvoke <java.net.InetAddress: java.net.InetAddress getLocalHost()>()

r0.<com.free.vpn.unblock.sites.proxybrowser.m.o: java.net.InetAddress a> = $r2

r321 = new com.free.vpn.unblock.sites.proxybrowser.m.p

$r7 = r0.<com.free.vpn.unblock.sites.proxybrowser.m.o: java.net.InetAddress a>

specialinvoke r321.<com.free.vpn.unblock.sites.proxybrowser.m.p: void <init>(com.free.vpn.unblock.sites.proxybrowser.m.o,java.net.InetAddress,int)>(r0, $r7, 8080)

virtualinvoke $r10.<java.net.ServerSocket: void bind(java.net.SocketAddress)>(r321)

Parameter

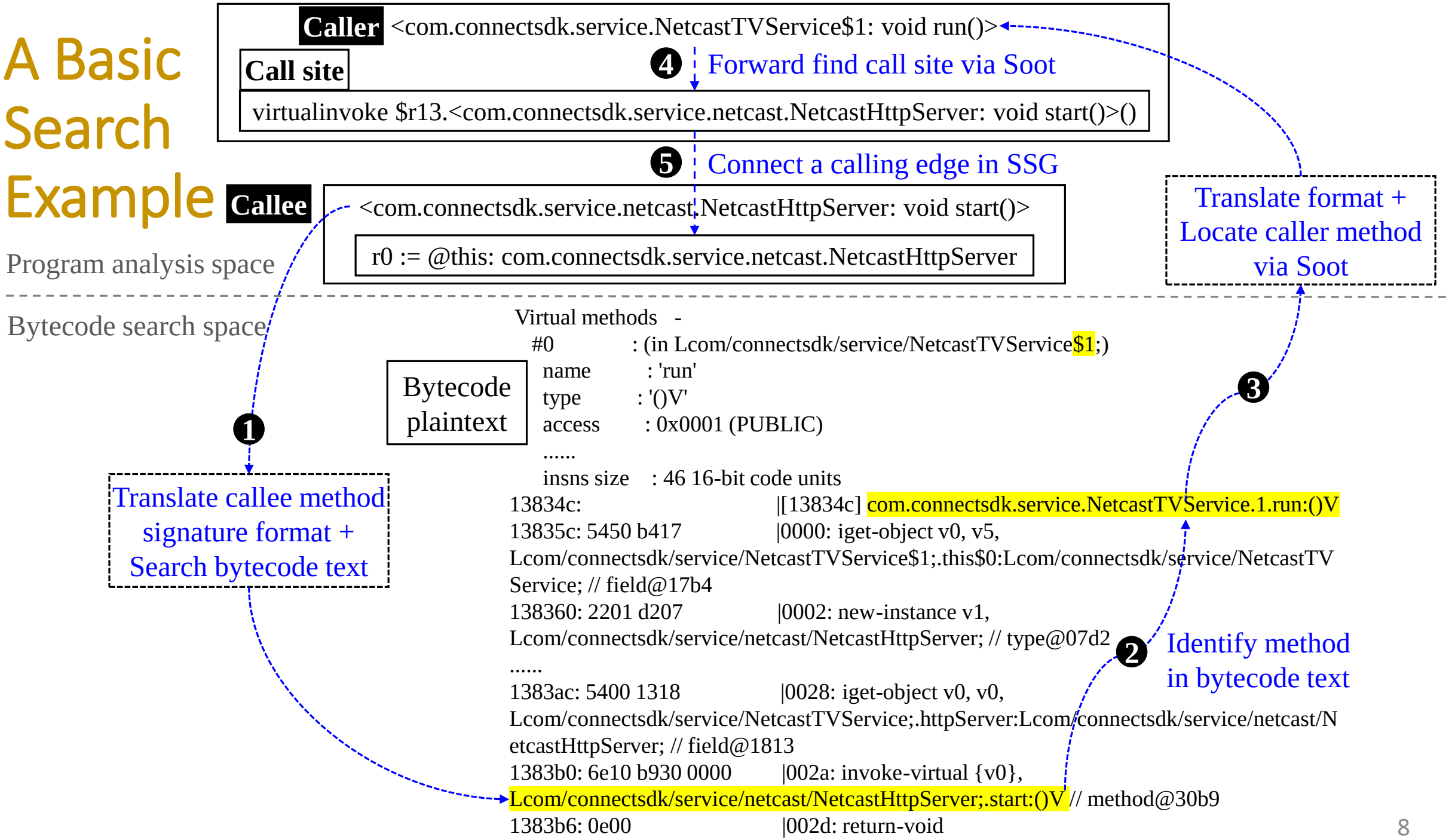
Sink API

On-the-fly bytecode search

Backward taint analysis

Forward Constant Propagation

A Basic Search Example



Bytecode Searches in Reality (pls refer to paper for details)

- The (basic) method signature-based search:
  - For static, private, and constructor callee methods that have only one signature
  - Work well for child methods by launching one more search with the child class
- Advanced search with forward object taint analysis:
  - For complex situations with Java polymorphism (super classes and interfaces), callbacks (e.g., onClick), and asynchronous flows (e.g., AsyncTask.execute).
  - First search the callee class's object constructors, and perform forward object taint analysis until reaching caller sites with the tainted object propagated into.
- Several special search mechanisms:
  - A recursive search for static initializers (i.e., static <clinit>() methods)
  - A two-time ICC search for inter-component communication (intent-related ICC)
  - An on-demand search for Android lifecycle handlers (e.g., onStart and onResume)

Experimental Setup

Dataset

- First crawl a set of **3,178** modern popular apps.
- Then search for the apps with all crypto and SSL sink APIs: **144** apps.

Environment

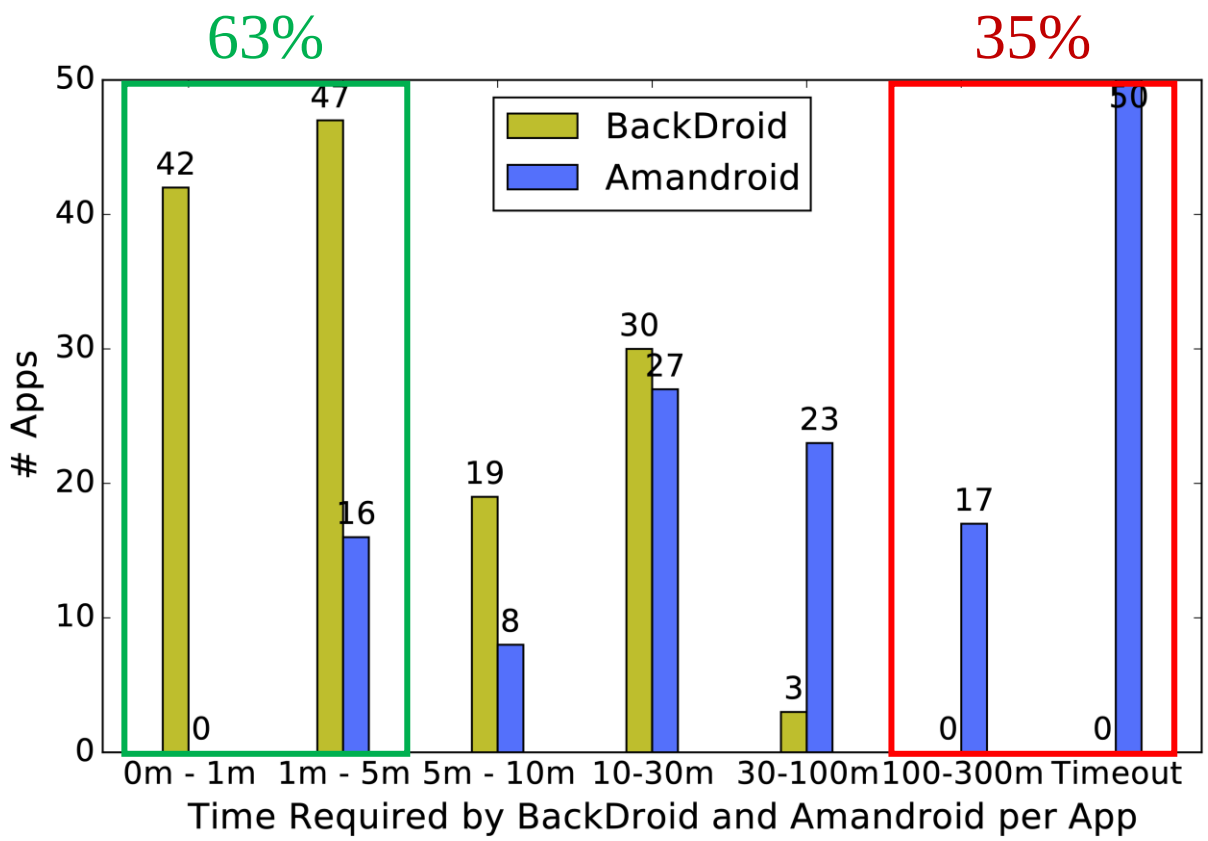
- Intel i7-4790 CPU (3.6GHZ, 8 cores);
- 16GB of physical memory;
- VM heap space: **12GB** for Amandroid **4GB** for BackDroid.

Tool configuration

- The **default** Amandroid configuration:
  - (per-component) timeout = **2m**
  - third_party_lib_file = /**liblist.txt**
- Per-app timeout: **300m** (or 5 hours)

Performance and Detection Results

- Performance:
  - 37 times faster (2.13min vs. 78.15min)
  - FlowDroid's CG time was 9.76min



- Maintain close detection effectiveness for the 30 vulnerable apps detected by Amandroid:
  - Uncovered 22 of 24 true positives and avoided six false positives
- 54 additional apps with potentially insecure crypto 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.

Thank You!

- BackDroid is open-sourced at https://github.com/VPRLab/BackDroid.

- We are cleaning and refactoring the code of BackDroid to make it easy-to-use and extensible.

- We are also evolving BackDroid so that it can be used as **a generic SDK** to support customization for different problems.

Questions?