# RunDroid: Recovering Execution Call Graphs for Android Applications

Yujie Yuan
East China Normal University
Shanghai, China
mijackstudio@gmail.com

Lihua Xu
East China Normal University
Shanghai, China
lhxu@cs.ecnu.edu.cn

Xusheng Xiao
Case Western Reserve University
Ohio, USA
xusheng.xiao@case.edu

Andy Podgurski
Case Western Reserve University
Ohio, USA
hap@case.edu

Huibiao Zhu
East China Normal University
Shanghai, China
hbzhu@sei.ecnu.edu.cn

## ABSTRACT

Fault localization is a well-received technique for helping developers to identify faulty statements of a program. Research has shown that the coverages of faulty statements and its predecessors in program dependence graph are important for effective fault localization. However, app executions in Android split into segments in different components, i.e., methods, threads, and processes, posing challenges for traditional program dependence computation, and in turn rendering fault localization less effective. We present RunDroid, a tool for recovering the dynamic call graphs of app executions in Android, assisting existing tools for more precise program dependence computation. For each execution, RunDroid captures and recovers method calls from not only the application layer, but also between applications and the Android framework. Moreover, to deal with the widely adopted multi-threaded communications in Android applications, RunDroid also captures methods calls that are split among threads.

Demo : https://github.com/MiJack/RunDroid
Video : https://youtu.be/EM7TJbE-Oaw

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

## KEYWORDS

Android, software analysis, multi-thread

**ACM Reference format:**
Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu. 2017. RunDroid: Recovering Execution Call Graphs for Android Applications. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122821


*ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany*

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122821

## 1 INTRODUCTION

Android applications (i.e., apps) have been witnessed a massive growth over the last decade. As such, more and more attentions are paid to the quality and reliability of apps. Among techniques that ensure high quality of apps, fault localization technique is a well-received technique for finding faulty statements of apps[12] [11]. Research [7, 8] has shown that the coverage of faulty statements and its predecessors in program dependence graph are important for effective fault localization. However, app executions split into segments in different components, i.e., methods, threads, and processes, posing challenges for traditional program dependence computation, and in turn rendering fault localization less effective. Specifically, the major challenges posed by Android apps for precise computation of program dependencies are as follows.

- *Multi-thread communications.* Android apps employ worker threads for intensive operations, while only objects running on the UI thread have access to UI objects. Hence, handlers are utilized commonly to pass messages and data between the UI thread and worker threads.
- *Implicit callbacks.* Android apps are driven by events and callbacks, such as onClick(), onActivityResult(), that are invoked by the Android framework.
- *Lifecycle methods.* In Android apps, each component, e.g. an activity, is required to follow a lifecycle, which are defined via callbacks.[1]

Despite achievements made by static analysis [6, 10, 14, 15] to model Android execution environment, these techniques still face challenges in precisely inferring the whole-program control flows of Android apps. Besides the aforementioned major challenges, Android apps use reflections (e.g., composing UI views and instantiation of components), type polymorphism (e.g., customized Thread classes and UI views), and temporary classes (e.g., event handlers registered using temporal classes), which cause further difficulties for static analysis to be precise and scalable.

To address these challenges and improve the precision of existing fault localization techniques for Android apps, we present RunDroid, a tool that recovers app executions' dynamic call graphs

[1] In this work, we specifically differentiate lifecycle methods from other implicit callback methods to better understand and visualize the internal behaviors of each activity.

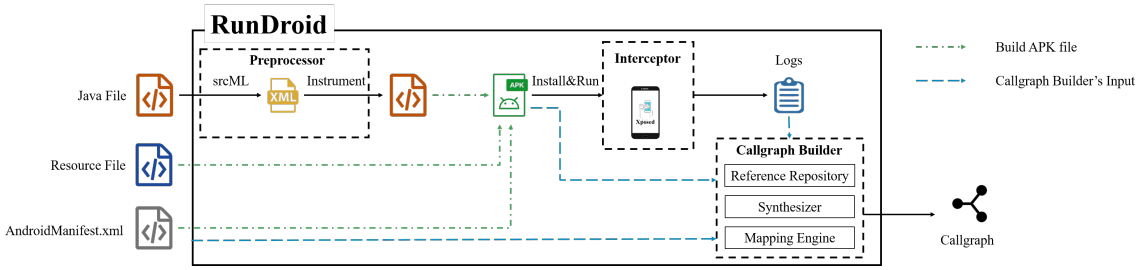



**Figure 1: Overview of RunDroid**

for constructing more precise program dependence. RunDroid instruments the program to capture method calls at the application layer, and intercepts messages between the application and the Android framework, including lifecycle methods. All the captured runtime information is then written to log files. For each execution, RunDroid identifies the captured method calls from the log files, and recovers the caller-callee relationships among them to construct a dynamic call graph. Furthermore, RunDroid captures the asynchronous implicit method calls introduced by Android's multi-threaded handler mechanism, and integrates them into the dynamic call graph to fill in the missing links between the thread initiating the asynchronous thread and the initiated asynchronous handler thread.

Additionally, we understand that human effort is typically unavoidable during fault localization. Developers will need to examine not only the estimated results and faulty statements, but also the execution traces with their data flows, especially for the failing test cases. RunDroid hence visualizes the recovered execution call graph, with the object information, to provide graphical assistance. To some extent, it can reduce the cost for tracking the execution situation for Android apps.

## 2 MOTIVATING EXAMPLE

We first use an example to intuitively explain the challenges faced by the traditional fault localization techniques. We then describe the major components of the RunDroid framework.

**Table 1: Motivating Example**

| | | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | |
|---|---|---|---|---|---|---|---|
| | v | btn0 | btn1 | btn1 | btn2 | btn1 | result |
| | num | - | -1 | 0 | - | -2 | $\tau$ |
| 1 | void onClick(View v) { | | | | | | |
| 2 | num = getNumber(); | 1 | 1 | 1 | 1 | 1 | NA |
| 3 | if(v.getId() == R.id.btn1) { | 1 | 1 | 1 | 1 | 1 | NA |
| 4 | if( num == 0 ) { | 0 | 1 | 1 | 0 | 1 | 0.67 |
| 5 | num=1; | 0 | 0 | 1 | 0 | 0 | -1.0 |
| 6 | } | | | | | | |
| 7 | } | | | | | | |
| 8 | Thread t = createThread(v.getId()); | 1 | 1 | 1 | 1 | 1 | NA |
| 9 | t.start(); | 1 | 1 | 1 | 1 | 1 | 0.67 |
| 10 | } | | | | | | |
| 11 | TaskThread.run() { | | | | | | |
| 12 | if(v.getId() == R.id.btn1) { | 1 | 1 | 1 | 1 | 1 | NA |
| 13 | **loadData(num);** /* **FAULT** */ | 0 | 1 | 1 | 0 | 1 | 0.67 |
| 14 | } | | | | | | |
| 15 | } | | | | | | |
| | | 0 | 1 | 0 | 0 | 1 | |

**Note:** The last columns show the estimates based on the causal influence model introduced in [7, 8].

In Table 1, we show an example faulty program with a faulty statement at Line 13. Instead being passed directly, the parameter of method *loadData(int)* at Line 13 should be checked if it is a positive value before passing into the method. The first column shows the program with line numbers associated with each of its statements. Columns 2 through 6 represent test cases $t_1$-$t_5$, respectively. The header of each test case column shows the values of $v$ and *num* that are used at Lines 1 and 2, respectively. The values for a test case column indicate whether the corresponding program statement is exercised by the test case, 1 for covered and 0 for not covered. The bottom row shows the outcome of each test case execution, with "1" indicating a failing execution and "0" indicating a passing one.

Suppose we compute the failure-causing effect of the program using the causal influence model [7, 8], we are able to compute the estimate numbers shown in Column "result". Lines 4, 9, and 13 are all estimated with the highest score 0.67.

Causal influence model, as well as many other fault localization techniques, considers the effects of dynamic program dependence, as dependence information contributes greatly to not only triggering the effects of faults, but also propagating them to program output. Unfortunately, due to Android's specific programming paradigm, the control dependency between Line 8 and Line 11 cannot be captured in the the example showed. And, losing appropriate control flows causes the computation to report three statements having the same highest score, rendering the fault location techniques less effective in Android. Therefore, to improve the fault localization techniques for Android apps, RunDroid aims to recover the very much needed control-flow dependencies through the runtime information and the dynamic call graphs during the executions.

## 3 RUNDROID

Figure 1 shows the overview of RunDroid. RunDroid takes the source code of an app as input, instruments the source code, and intercepts the executions of the instrumented app to analyze message objects. After each execution, RunDroid produces a set of log files, which will be further analyzed to generate the dynamic call graph for the execution.

RunDroid consists of three components. The *preprocessor* component instruments the program source to probe instructions, so that every method invoked during execution will be properly logged.

The second component is an *interceptor* to capture the method calls between application layer and the Android framework, including lifecycle methods and the implicit callbacks. It predefines a set of method calls of interest and logs their executions. Take class *android.app.Activity* as an example, the methods *setContentView(android.view.View)* and *onCreate(android.os.Bundle)* are the methods of interest. The invocations of these methods usually mean that the

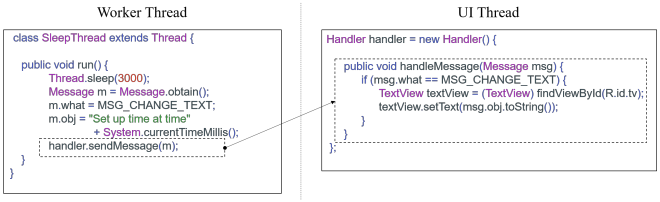



**Figure 2: The snippet code about Handler**

messages are transmitted between the application and the Android framework. Whenever these methods are invoked during execution, RunDroid intercepts the messages, and associates them with the corresponding method calls in the application layer to produce a complete execution call trace.

The *callgraph builder*, shown on the right in Figure 1, consists of an *reference repository*, a *mapping engine*, and a *synthesizer*. The callgraphs builder analyzes the log files produced by the first two components, identifies the multi-threaded implicit asynchronous executions, fills in the missing links between threads, and outputs the complete method call graph for each execution. The complete method call graph can be used to construct more precise program dependence.

## 3.1 Capturing Application Layer Method Calls

To capture the method calls in application layer, we use program instrumentation to log the runtime execution trace. There are a number of automatic tools for instrumentation on the source or bytecode of Java programs[1]. Since the context of utilizing RunDroid is during debugging phase, it is natural to instrument the program at source level. Additionally, we observe that bytecode instrumentation techniques, such as the one Emma[3] uses, suffer from the 64K reference limit[2]. Hence, the *Preprocessor* component builds upon srcML [9], that converts Java source code into xml-format, so that the methods are monitored during execution.

## 3.2 Recovering Method Calls between Application and the Android Framework

The *Interceptor* component is built upon Xposed framework [5], which intercepts messages passing between the application layer and the Android framework. Interceptor logs each method calls made between the two layers and associates them with the corresponding method calls in application layer to produce a complete method call trace. RunDroid maintains a list of methods that are of interest, such as lifecycle methods and implicit callbacks, so that the log files contain the method calls invoked during each execution.

## 3.3 Building Dynamic Call Graphs

In Android, whenever an application is launched, the system creates a UI thread as the main thread. UI thread is responsible for drawing the user interface including dispatching events to corresponding UI widgets. To avoid blocking event dispatching and laggy responses at the user interface, long-running tasks such as network accesses, complicated processing, and others are done via worker threads. Handlers are then used to pass messages between the UI threads and the worker threads. Take the code snippet at Figure 2 as an example, when the worker thread (left) needs to pass data to the UI thread (right), we can invoke the methods of *handler* class to send or receive messages. Hence, there is an implicit asynchronous invocation between the method *sendMessage()* at the worker thread and the method *handleMessage()* at the UI thread.

Such implicit asynchronous invocations can easily cause static analysis to explode in matching methods that send and receive messages, and produce large number of false positives. As such, RunDroid addresses this challenge by matching these invocations through dynamic execution traces, which results in the development of three sub-components: *Reference Repository* component, *Mapping Engine* component, *Synthesizer* component. The *Reference Repository* component, built upon Neo4j[4], processes the log files and stores the captured methods and their execution calls. The *Mapping Engine* component, which is built upon soot [13], identifies asynchronous invocation pairs from the log history, searches for their corresponding instantiation classes, and stores these captured multi-threaded method calls as "trigger" relation. The *Synthesizer* component then integrates these trigger relations with the captured execution calls into a complete execution call trace, and stores at the *Reference Repository*.

# 4 EVALUATION

To illustrate how the dynamic call graphs built by RunDroid assists fault localization techniques, we compare the estimation results using the causal influence model with or without RunDroid. We also present another case study with more complicated program dependencies for demonstrating RunDroid's visualization of dynamic call graph and data flows, providing visual assistance for fault localization.

## 4.1 Effectiveness Study

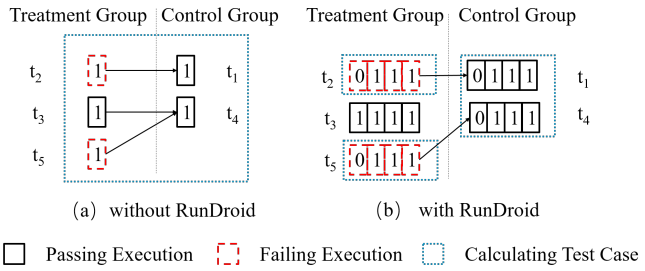
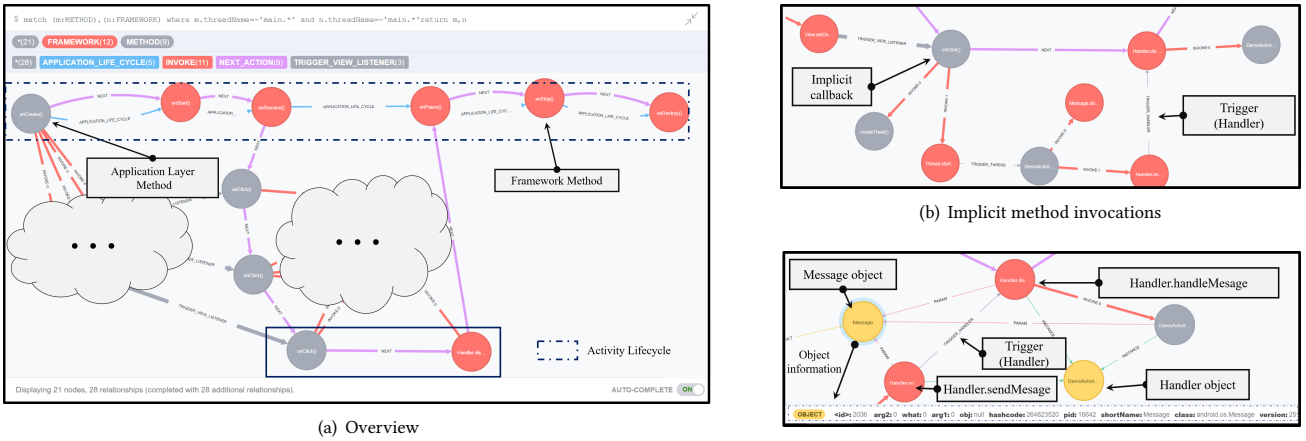



**Figure 3: Units in treatment group and control group with their covariate values**

To compare the effectiveness of causal influence model with and without the dynamic call graphs provided by RunDroid, we follow the experiment setup at [7, 8], use the original causal influence model (without RunDroid) as a baseline, and compare their computed *causal-effect estimates*.

We take the motivating example as the subject program. Causal influence model relies heavily on dependence relationship among statements. With RunDroid, the asynchronized trigger relation between Line 9 *t.start()* and Line 11 *TaskThread.run()*, is captured, hence the predecessors of Line 13 are Lines {5, 8, 9, 13}, instead of Line 13 only, which is the case of original causal influence model.




(a) Overview

(b) Implicit method invocations

(c) Related object information Box

**Figure 4: The RunDroid Output**

Figure 3 shows the treatment units and the control units with their associated test cases for Line 13. Specifically, Figure 3-(a) represents the case of without RunDroid, and Figure 3-(b) represents the case with RunDroid. The treatment units, which are test cases that cover Line 13 are $t_2$, $t_3$, and $t_5$; the control units, which are test cases that do not cover Line 13 are $t_1$ and $t_4$. For target statement, we use the vector of covariate to present the cover information of its predecessor statements in a test case. In Figure 3-(b), the vector $\langle 0,1,1,1\rangle$ means that for test case $t_2$, only Line 5 is not covered, the other three predecessor statements (Lines 8, 9, 13) are all covered.

Using the same equation from causal inference model, the causal-effect estimates for the latter case (with dependence information from RunDroid) are resulted differently. The estimate for Line 13 is $\frac{(1+1)}{2} - \frac{(0+0)}{2} = 1.0$. The estimates for the other statements are shown in Table 2, in the far right column.

**Table 2: Comparing Results**

| | | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $\tau$ | $\tau'$ |
|---|---|---|---|---|---|---|---|---|
| 1 | void onClick(View v) { | | | | | | | |
| 2 | num = getNumber(); | 1 | 1 | 1 | 1 | 1 | NA | NA |
| 3 | if(v.getId() == R.id.btn1) { | 1 | 1 | 1 | 1 | 1 | NA | NA |
| 4 | if( num == 0 ) { | 0 | 1 | 1 | 0 | 1 | 0.67 | 0.67 |
| 5 | num=1; | 0 | 0 | 1 | 0 | 0 | -1.0 | -1 |
| 6 | } | | | | | | | |
| 7 | } | | | | | | | |
| 8 | Thread t = createThread(v.getId()); | 1 | 1 | 1 | 1 | 1 | NA | NA |
| 9 | t.start(); | 1 | 1 | 1 | 1 | 1 | 0.67 | 0.67 |
| 10 | } | | | | | | | |
| 11 | TaskThread.run() { | | | | | | | |
| 12 | if(v.getId() == R.id.btn1) { | 1 | 1 | 1 | 1 | 1 | NA | 0.67 |
| 13 | **loadData(num); /* FAULT */** | 0 | 1 | 1 | 0 | 1 | 0.67 | 1 |
| 14 | } | | | | | | | |
| 15 | } | | | | | | | |
| | | 0 | 1 | 0 | 0 | 1 | | |

## 4.2 Case Study of RunDroid's Visualization

As mentioned, we understand that manual examination of data flows during fault localization is unavoidable, RunDroid hence provides visualization for: (1) The exact event sequences for each test execution, instead of all possible ones as the static analyzers do; (2) The execution call graphs, that are typically hard to capture with static analyzers; (3) Related object information, that assist in data flow analysis of static analyzers.

To demonstrate the effectiveness of RunDroid's visualization capability, we show here yet another case study with more complicated program dependencies. Figure 4 shows the three types of outputs from RunDroid. Figure 4-(a) highlights the overview of event sequences initiated at the app's UI thread; The captured events are represented as nodes, and the purple link between nodes represents their relationship, which denoted as "NEXT_ACTION". The color of nodes denotes if the method is defined at application layer or the framework layer, labled as METHOD or FRAMEWORK, respectively. The Activity lifecycle methods are shown at the upper part inside the dotted box. We zoom in the bottom box and show it as Figure 4-(b), which shows the detailed execution calls. Here we show the implicit method invocations, such as callbacks and multi-threaded communications, that are captured during each execution. RunDroid denotes application layer method calls or intercepted method calls between application layer and framework as "INVOKE" relations, and the implicit multi-threaded asynchronized method calls as "TRIGGER" relations. To assist in data flow analysis, RunDroid also captures and visualizes the object information related to the methods invocations. Take the "TRIGGER" relation shown at Figure 4-(b) as example and zoom in further, RunDroid displays the related object information as Figure 4-(c).

## 5 CONCLUSIONS

In this paper, we have presented RunDroid, a tool that captures the dynamic program dependences during each app execution, and recovers the complete dynamic call graph. RunDroid is intended to serve as a complementary to existing tools and techniques, including static analysis tools and fault localization techniques; With visualized execution history and more precise program dependence information for each app execution, Rundroid assists developers and tools with more precise causal-effect estimates for fault localization.

## ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China, under Grants 61502170 and in part by Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things(No. ZF1213).

## REFERENCES


[1] 2017. The AspectJ Project. (2017). Retrieved May 15,2017 from https://eclipse.org/aspectj/

[2] 2017. Configure Apps with Over 64K Methods | Android Studio. (2017). Retrieved May 15,2017 from https://developer.android.com/studio/build/multidex.html

[3] 2017. EMMA SDK integration for Android EMMA Support. (2017). Retrieved May 15,2017 from http://support.emma.io/hc/en-us/sections/201193992-EMMA-SDK-integration-for-Android

[4] 2017. Neo4j, the world's leading graph database - Neo4j Graph Database. (2017). Retrieved May 24,2017 from https://neo4j.com/

[5] 2017. Xposed. (2017). Retrieved May 15,2017 from http://repo.xposed.info/module/de.robv.android.xposed.installer

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[7] George K Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 73–84.

[8] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*.

[9] ML Collard, M Decker, and JI Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code. In *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSMâĂŹ13) Tool Demonstration Track, Eindhoven, The Netherlands*. 1–4.

[10] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*.

[11] Pedro Machado, José Campos, and Rui Abreu. 2013. MZoltar: automatic debugging of Android applications. In *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. ACM, 9–16.

[12] Hamed Mirzaei and Abbas Heydarnoori. 2015. Exception fault localization in Android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. IEEE, 156–157.

[13] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

[14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.

[15] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 89–99.