A Whitebox Approach for Automated Security Testing of
自动化安全测试的白盒方法
Android Applications on the Cloud
云端 Android 应用程序

Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, Angelos Stavrou
Riyadh Mahmood Naeem Esfahani Thabet Kacem Nariman Mirzaei Sam Malek Angelos Stavrou
Computer Science Department
计算机科学系
George Mason University
乔治梅森大学
{rmahmoo2, nesfaha2, tkacem, nmirzaei, smalek, astavrou}@gmu.edu
{ rmahmoo2 nesfaha2 tkacem nmirzaei smalek astavrou }@gmu. edu

***Abstract*— By changing the way software is delivered to end-users, markets for mobile apps create a false sense of security: apps are downloaded from a market that can potentially be regulated. In practice, this is far from truth and instead, there has been evidence that security is not one of the primary design tenets for the mobile app stores. Recent studies have indicated mobile markets are harboring apps that are either malicious or vulnerable leading to compromises of millions of devices. The key technical obstacle for the organizations overseeing these markets is the lack of practical and automated mechanisms to assess the security of mobile apps, given that thousands of apps are added and updated on a daily basis. In this paper, we provide an overview of a multi-faceted project targeted at automatically testing the security and robustness of Android apps in a scalable manner. We describe an Android-specific program analysis technique capable of generating a large number of test cases for fuzzing an app, as well as a test bed that given the generated test cases, executes them in parallel on numerous emulated Androids running on the cloud.**

摘要——通过改变软件交付给终端用户的方式，移动应用程序市场制造了一种虚假的安全感: 应用程序是从一个可能受到监管的市场下载的。实际上，这远非事实，相反，有证据表明，安全性并不是移动应用商店的主要设计原则之一。最近的研究表明，移动市场上存在着恶意或易受攻击的应用程序，这些应用程序会危及数以百万计的设备。监督这些市场的组织面临的主要技术障碍是缺乏实用和自动化的机制来评估移动应用程序的安全性，因为每天都有成千上万的应用程序被添加和更新。在本文中，我们提供了一个多方面项目的概述，这个项目旨在以可扩展的方式自动测试 Android 应用程序的安全性和健壮性。我们描述了一种特定于 android 的程序分析技术，它能够生成大量用于模糊应用程序的测试用例，以及一个给定生成的测试用例的测试平台，这些测试用例在运行在云上的大量仿真 android 上并行执行。

***Keywords-*Android; Security Testing; Program Analysis**
Android; 安全测试; 程序分析

I. Introduction
引言

Mobile App markets are creating a fundamental paradigm shift in the way software is delivered to the end users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software development field, allowing small entrepreneurs to compete head-to-head against prominent software development companies. The result of this has been an explosive growth in the number of new apps for platforms, such as Mac, Android, and iPhone, that have embraced this model.

移动应用市场在软件交付给终端用户的方式上正在创造一个根本性的范式转变。这种软件供应模式的好处很多，包括能够快速有效地获取、引入、维护和增强消费者使用的软件。通过提供一个以名义成本进入大型消费者市场的媒介，应用程序市场已经平衡了软件开发领域，允许小企业家与知名软件开发公司进行正面竞争。其结果是，支持这种模式的 Mac、 Android 和 iPhone 等平台的新应用程序数量出现了爆炸性增长。

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we are witnessing an increase in the security threats targeted at platforms that have embraced this paradigm. This is nowhere more evident than in the *Android market*, where many cases of apps infected with malwares and spywares have been reported [1]. Numerous culprits are in play here, and some are not even technical, such as the general lack of an overseeing authority in the case of open markets and inconsequential implication to those caught provisioning applications with vulnerabilities or malicious capabilities.

然而，这种范式转变引起了一系列新的安全挑战。随着应用程序市场的出现，我们目睹了针对采用这种模式的平台的安全威胁的增加。这在 Android 市场中最为明显，在那里已经报道了许多应用程序感染恶意软件和间谍软件的案例[1]。这里有许多罪魁祸首，其中一些甚至不是技术性的，例如在开放市场的情况下普遍缺乏监督机构，对于那些被发现提供具有漏洞或恶意能力的应用程序的人来说，影响微不足道。

From a technical standpoint, however, the key obstacle is the lack of practical techniques to rapidly assess and test the

然而，从技术的角度来看，关键的障碍是缺乏实用的技术来快速评估和测试

security of applications submitted to the market. Security testing is generally a manual, expensive, and cumbersome process. This is precisely the challenge that we have begun to address through the development of a framework that aids the analysts in testing the security of Android apps. The framework is comprised of a tool-suite that given an app automatically generates and executes numerous test cases, and provides a report of uncovered security vulnerabilities to the human analyst. We have focused our research on Android as (1) it provides one of the most widely-used and at the same time vulnerable app markets, (2) it dominates the smartphone consumer market, and (3) it is open-source, lending itself naturally for experimentation in the laboratory.
向市场提交的申请的安全性。安全测试通常是一个手工的、昂贵的、繁琐的过程。这正是我们通过开发一个框架来帮助分析师测试 Android 应用程序的安全性所面临的挑战。该框架由一个工具套件组成，该工具套件给定一个应用程序自动生成并执行大量测试用例，并向人工分析师提供一份未发现的安全漏洞报告。我们将研究重点放在 Android 上，因为(1)它提供了应用最广泛、同时也是最脆弱的应用程序市场之一; (2)它主导着智能手机消费市场; (3)它是开源的，自然而然地为实验室的实验提供了条件。

Security testing is a notoriously difficult task. This is partly because unlike functional testing that aims to show a software system complies with its specification, security testing is a form of *negative testing*, i.e., showing that a certain (often apriori unknown) behavior does not exist.

众所周知，安全测试是一项艰巨的任务。这在一定程度上是因为，与旨在表明软件系统符合其规范的功能测试不同，安全性测试是一种否定测试，即表明某种(通常是先验未知的)行为并不存在。

A form of security testing that does not require test case specification or significant upfront effort is *fuzz testing*, or simply *fuzzing* [2]. In short, fuzzing is a form of negative testing that feeds malformed and unexpected input data to a program with the objective of revealing security vulnerabilities. Programs that are used to create and examine fuzz tests are called *fuzzers*. Fuzzers have been employed in the past by the hacking community as one of the predominant ways of breaking into a system [2]. For instance, an SMS protocol fuzzer [3] was recently shown to be highly effective in finding severe security vulnerabilities in all three major smartphone platforms. In the case of Android, fuzzing found a security vulnerability triggered by simply receiving a particular type of SMS message, which not only kills the phone's telephony process, but also kicks the target device off the network [3].

一种不需要测试用例规范或大量前期工作的安全测试形式是模糊测试，或者简单的模糊测试[2]。简而言之，模糊测试是一种负面测试形式，它将错误的和意想不到的输入数据反馈给程序，目的是揭示安全漏洞。用于创建和检查模糊测试的程序被称为模糊测试程序。模糊程序在过去被黑客社区用作入侵系统的主要方式之一[2]。例如，短信协议模糊程序[3]最近被证明在所有三个主要智能手机平台上发现严重的安全漏洞非常有效。在 Android 的例子中，fuzzing 发现了一个安全漏洞，只需要接收一种特定类型的短信即可触发安全漏洞，这不仅会杀死手机的电话进程，而且还会将目标设备踢出网络[3]。

Despite the individual success of fuzzing as a general method of identifying vulnerabilities, fuzzing has traditionally been used as a brute-force mechanism. There has been a lack of sophisticated or guided techniques for fuzz testing apps, in particular those targeted at smartphone platforms. Using fuzzing for testing is generally a time consuming and computationally expensive process, as the space of possible inputs to any real-world program is often unbounded. Existing fuzzing tools, such as Android's Monkey [4], generate purely random test case inputs, and thus are often ineffective in practice.

尽管模糊化作为识别漏洞的一种通用方法取得了个人成功，但模糊化传统上被用作一种蛮力机制。对于模糊测试应用程序，尤其是针对智能手机平台的应用程序，一直缺乏复杂的或指导性的技术。使用模糊进行测试通常是一个耗时且计算成本高昂的过程，因为任何真实程序的可能输入空间往往是无限的。现有的模糊化工具，如 Android 的 Monkey [4]，产生纯粹随机的测试用例输入，因此在实践中往往是无效的。

In this paper, we are addressing these limitations by developing a <u>scalable</u> approach for <u>intelligent</u> fuzz testing of Android applications. Our approach *scales* in terms of code size and number of test cases. We achieve that by leveraging

在本文中，我们通过开发一种可伸缩的 Android 应用程序智能模糊测试方法来解决这些局限性。我们的方法根据代码大小和测试用例的数量进行扩展。我们通过充分利用

the unprecedented computational power of cloud computing. The framework employes numerous heuristics and software analysis techniques to *intelligently* guide the generation of test cases aiming to boost the likelihood of discovering vulnerabilities. The proposed testing mechanisms empower the broader app market community to harness the immense computational power of cloud together with novel automated testing techniques to quickly, accurately, and cheaply find security vulnerabilities.

云计算前所未有的计算能力。该框架采用了大量的启发式和软件分析技术来智能地指导测试用例的生成，以提高发现漏洞的可能性。提出的测试机制使更广泛的应用程序市场社区能够利用云计算的巨大计算能力，结合新颖的自动化测试技术，快速、准确、廉价地发现安全漏洞。

This paper is organized as follows. Section II provides a background on Android and its security model. Section III outlines an Android app that is used for illustrating the research. Section IV provides an overview of our approach, while Sections V to IX provide the details. The paper concludes with an overview of the related research in Section X and a discussion of our future work in Section XI.

这篇论文的组织方式如下。第二部分提供了 Android 及其安全模型的背景知识。第三部分概述了一个用于说明研究的 Android 应用程序。第四部分概述了我们的方法，第五至第九部分提供了细节。本文最后对第十部分的相关研究进行了概述，并对第十部分的未来工作进行了讨论。

II. Background
背景

In 2008, Google and Open Handset Alliance launched Android Platform for mobile devices. Android is a comprehensive software framework for mobile communication devices including smart-phones and PDAs.

2008 年，谷歌和开放手机联盟推出了面向移动设备的 Android 平台。Android 是一个全面的移动通信设备软件框架，包括智能手机和 pda。

A. Android Architecture and Security Model
安卓架构和安全模型

The Android framework includes a full Linux operating system based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. Google Android platform is based on *Dalvik Virtual Machine (DVM)*

Android 框架包括基于 ARM 处理器的完整 Linux 操作系统、系统库、中间件和一套预安装的应用程序。Google Android 平台基于 Dalvik Virtual Machine (DVM)

[5] for executing and containing programs written in Java. Android also comes with an *application framework*, which provides a platform for application development and includes services for building GUI applications, data access, and other component types. The framework is designed to simplify the reuse and integration of components. Applications publish their capabilities and others can use them, subject to security constraints described further below.

用于执行和包含用 Java 编写的程序。Android 还提供了一个应用程序框架，它为应用程序开发提供了一个平台，并包括用于构建 GUI 应用程序、数据访问和其他组件类型的服务。该框架旨在简化组件的重用和集成。应用程序发布它们的功能，其他人可以使用它们，受到下面进一步描述的安全约束。

Android enforces its application security mechanisms at two levels. The first level of security is achieved by forcing each application to execute within its own *secure sandbox,* which sets Android apart from other operating systems present in the market. Thus, an instance of an application is isolated from other applications in the memory.

Android 在两个层次上强制执行其应用程序安全机制。第一级安全性是通过强制每个应用程序在自己的安全沙盒中执行来实现的，这将 Android 与市场上的其他操作系统区分开来。因此，一个应用程序的实例与内存中的其他应用程序是隔离的。

A second level of security enforcement is achieved through Android’s permission based security model. Android uses Mandatory Access Control to regulate access to applications and phone resources based on access permission policies. These policies are implemented in the form of permission labels that are assigned to components and applications. The permissions granted to each application are defined in its mandatory *manifest* file. The *manifest* file values are bound to the application at compile time and cannot be changed afterwards unless the application is recompiled.

第二级安全实施是通过 Android 的基于权限的安全模型实现的。Android 使用强制访问控制来根据访问权限政策来管理对应用程序和手机资源的访问。这些策略以分配给组件和应用程序的权限标签的形式实现。授予每个应用程序的权限在其强制清单文件中定义。清单文件值在编译时绑定到应用程序，除非应用程序被重新编译，否则不能在编译后更改。

Android’s security mechanisms have been breached numerous times in the past [1]. In general, once the user installs an application infected with a malware, not much protection can be achieved through Android’s standard security mechanisms. Since the access permissions in Android are coarse-grained (i.e., high-level all or nothing

Android 的安全机制在过去曾多次被攻破[1]。一般来说，一旦用户安装了一个感染了恶意软件的应用程序，通过 Android 的标准安全机制就不能实现多少保护。因为 Android 的访问权限是粗粒度的(比如，高级的全有或全无)

permissions), a malware embedded in an application can use all of the access permissions granted to the host application. A malicious program can also quickly exhaust, or expose to remote attacks, important system resources. At the same time, unlike desktop computing, it is hard to employ a large, resource intensive program (e.g., Antivirus) to detect, monitor, and control malicious software. Finally, malwares and attackers often leverage bad implementation choices and unintentional bugs to realize their objectives [1].
权限)，嵌入在应用程序中的恶意软件可以使用授予主机应用程序的所有访问权限。一个恶意程序也可以迅速耗尽，或暴露于远程攻击，重要的系统资源。与此同时，与桌面计算不同的是，很难使用大型的资源密集型程序(例如，反病毒程序)来检测、监视和控制恶意软件。最后，恶意软件和攻击者经常利用错误的实现选择和无意的 bug 来实现他们的目标[1]。

B. Android Application Building Blocks
B. Android 应用程序构建模块

As mentioned earlier, each Android application has a mandatory *manifest* file. This is a required XML file for every application and provides essential information for managing the life cycle of an application to the Android platform. Examples of the kinds of information included in a manifest file are descriptions of the application's *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers* among other architectural and configuration properties.

正如前面提到的，每个 Android 应用程序都有一个强制性的清单文件。这是每个应用程序所必需的 XML 文件，并且为管理 Android 平台的应用程序的生命周期提供了必要的信息。清单文件中包含的信息类型的示例包括对应用程序的 Activities、 Services、 Broadcast receiver 和 Content provider 以及其他架构和配置属性的描述。

An *Activity* is a screen that is presented to the user and contains a set of layouts (e.g., *LinearLayout* that organizes items within the screen horizontally or vertically). The layouts contain GUI controls, known as *view widgets* (e.g., *TextView* for viewing text and *EditText* for text inputs). The layouts and its controls are usually described in a configuration XML file with each layout and control having a unique identifier. A *Service* is a component that runs in the background and performs long running tasks, such as playing music. Unlike an *Activity*, a *Service* does not present the user with a screen for interaction. A *Content Provider* manages structured data stored on the file system or database, such as contact information. A *Broadcast Receiver* responds to system wide announcement messages, such as the screen has turned off or the battery is low. *Activities*, *Services*, and *Broadcast Receivers* are activated via *Intent* messages. An *Intent* message is an event for an action to be performed along with the data that supports that action. *Intent* messaging allows for late run-time binding between components, where the calls are *not explicit* in the code, rather connected through event messaging.

Activity 是一个呈现给用户的屏幕，它包含一组布局(例如，LinearLayout 在屏幕上水平或垂直地组织项目)。布局包含 GUI 控件，称为视图部件(例如，用于查看文本的 TextView 和用于文本输入的 EditText)。布局及其控件通常在配置 XML 文件中描述，每个布局和控件都有一个唯一标识符。Service 是一个在后台运行的组件，它执行长时间运行的任务，比如播放音乐。与 Activity 不同，Service 不会为用户提供一个交互的屏幕。Content Provider 管理存储在文件系统或数据库中的结构化数据，比如联系人信息。Broadcast Receiver 响应系统范围内的公告消息，比如屏幕已关闭或电池电量不足。活动、服务和广播接收器通过意图消息被激活。Intent 消息是一个事件，一个操作和支持该操作的数据一起被执行。Intent 消息允许组件之间的后期运行时绑定，其中的调用在代码中不是显式的，而是通过事件消息连接。

Activity and *Service* are required to follow prespecified lifecycles [6]. For instance, Figure 1 shows the events in the lifecycle of an *Activity*: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, *onRestart()*, and *onDestroy()*. These lifecycle events play an important role in our research as explained later.

活动和服务需要遵循预先指定的生命周期[6]。例如，图 1 显示了 Activity: onCreate ()、 onStart ()、 onResume ()、 onPause ()、 onStop ()、 onRestart () 和 onDestroy () 的生命周期中的事件。这些生命周期事件在我们的研究中扮演着重要的角色。

In addition to these components, a typical application utilizes many resources. These resources include animation files, graphics files, layout files, menu files, string constants, styles for user interface controls. Most of these are described using XML files. An example, as mentioned before are layouts. The layout XML files define the architecture of user interface controls that are used by *Activities*. The resources each have a unique identifier that is used to distinguish and get a reference to them in the application code.

除了这些组件之外，典型的应用程序还使用许多资源。这些资源包括动画文件，图形文件，布局文件，菜单文件，字符串常量，用户界面控件的样式。其中大多数都使用 XML 文件进行描述。前面提到的一个例子是布局。布局 XML 文件定义了 Activities 使用的用户界面控件的架构。每个资源都有一个唯一标识符，用于在应用程序代码中区分和获取对它们的引用。

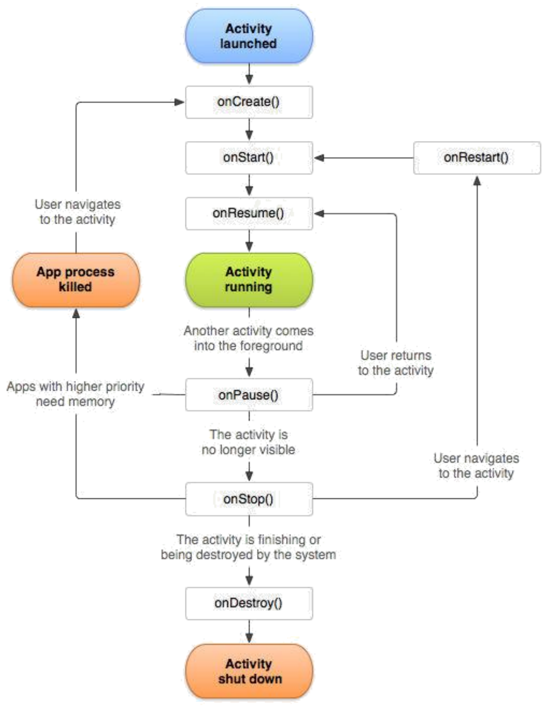

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1. Lifecycle of *Activity* in Android from [6].
图 1. 来自[6]的 Android 中 Activity 的生命周期。

III. Illustrative Example
说明性例子

We use a subset of a software system, called Emergency Deployment System (EDS) [7], to illustrate our research. EDS is a system previously developed in collaboration with a government agency for the deployment of personnel in emergency response scenarios. EDS is intended to allow a search and rescue crew to share and obtain an assessment of the situation in real-time (e.g., interactive overlay on maps), coordinate with one another (e.g., send reports, chat, and share video streams), and engage the headquarters (e.g., request resources).

我们使用一个软件系统的子集，称为紧急部署系统(EDS)[7]，来说明我们的研究。EDS 是以前与政府机构合作开发的一个系统，用于在紧急情况下部署人员。EDS 旨在让搜救人员实时分享和获取情况评估(例如，地图上的互动覆盖)，彼此协调(例如，发送报告、聊天和共享视频流)，以及与总部联系(例如，请求资源)。

EDS has several interrelated apps. One of them is a *Driving Direction* app that can be used to calculate off-road driving directions between two geographic points, while considering objectives such as distance, time, and safety. Figure 3a depicts the GUI for this app. This screen has four input text boxes and three buttons. The input boxes are for latitude/longitude pair and the buttons for alternative ways of

EDS 有几个相关的应用程序。其中之一是一个 Driving Direction 应用程序，可以用来计算两个地理点之间的越野行驶方向，同时考虑距离、时间和安全等目标。图 3a 描述了这个应用程序的图形用户界面。这个屏幕有四个输入文本框和三个按钮。输入框用于纬度/经度对，按钮用于其他方式

computing the
计算

directions. The latitude/longitude coordinates can be typed in or selected from a map. The resulting turn-by-turn directions are shown in a separate text box,
方向。纬度/经度坐标可以在地图上输入或选择。由此产生的逐个路径的方向显示在一个单独的文本框中,

and optionally displayed on a map.
并可选地显示在地图上。

IV. Approach Overview
IV. 方法概述

Figure 2 illustrates an overview of our approach. The parts of the approach depicted within a dotted bubble run on a cloud platform to allow for the execution of large number of test cases on many instances of a given application.

图 2 说明了我们方法的概述。在云平台上运行的点状气泡中描述的方法部分，允许在给定应用程序的许多实例上执行大量测试用例。

The input to our framework is an *Android application package file (APK)*. APKs are Java bytecode packages used to distribute and install Android apps. If the source code is not readily available, we first reverse engineer the APK file using one of the available tools for this purpose (e.g., *apktool* [8], *dex2jar* [9], *smali* [10], and *dedexer* [11]). We then leverage *JD-GUI* [12] for decompiling the Java class files to obtain the source files.

我们框架的输入是一个 Android 应用程序包文件 (APK)。APKs 是用于分发和安装 Android 应用程序的 Java 字节码包。如果源代码不容易获得，我们首先使用可用的工具(例如 apktool [8]、 dex2jar [9]、 smali [10]和 dedexer [11])对 APK 文件进行反向工程。然后我们利用 JD-GUI [12]来反编译 Java 类文件以获得源文件。

The first step is to discover the app's *Input Surface*, which corresponds to all the ways in which an application can be initiated or accessed. We use MoDisco [13] to parse the source code we obtained directly or via decompilation of the bytecode. In addition, we process the resources and configuration information including the *manifest* file. From the generated data, we automatically construct two models of the app: *Call Graph Model* and *Architectural Model*. We base our analysis on these two models. The *Call Graph Model* represents all possible method invocation sequences (execution traces) within an app. The *Architectural Model* represents the app's architecture and user interface layout constructed from the meta-data associated with Android apps (i.e., recall *manifest* and *layout* XML *files* from Section II).

第一步是发现应用程序的 Input Surface，它对应于启动或访问应用程序的所有方式。我们使用 MoDisco [13]来解析我们直接获得的源代码或通过反编译字节码获得的源代码。另外，我们处理包括清单文件在内的资源和配置信息。根据生成的数据，我们自动构建了两个应用程序模型：Call Graph Model 和 Architectural Model。我们的分析基于这两个模型。Call Graph 模型表示应用程序中所有可能的方法调用序列(执行轨迹)。Architecture Model 表示应用程序的架构和用户界面布局，这些布局是由与 Android 应用程序相关的元数据构建的(例如，从 Section II 中召回清单和布局 XML 文件)。

The *Test Case Generator* uses these models together with the Android specifications (for GUI controls, widgets, APIs, etc.) and template Android test case skeletons to generate test cases. A test case template is a skeleton Java file that contains the common static portions of a test case. For example, the *JUnit* methods such as *setUp()* and *tearDown()* come standard as part of the template.

Test Case Generator 使用这些模型以及 Android 规范(GUI 控件、小部件、 api 等)和模板 Android 测试用例框架来生成测试用例。测试用例模板是一个包含测试用例常见静态部分的 Java 框架文件。例如，JUnit 方法如 setUp ()和 tearDown ()作为模板的一部分是标准的。

Following the generation of test cases, the *Test Execution Environment* is activated to simultaneously execute the tests on numerous instances of the same application. For scalability, we harness the parallelism of a cloud-based system to execute the tests on virtual nodes running the *Android Emulator*. In addition to code coverage, several other Android-specific *Monitoring Facilities* such as *Intent Sniffer* [14] are instantiated and deployed to collect runtime data as tests execute. These monitoring facilities record program behavior and errors (e.g., crashes, exceptions, access violations, resource thrashing) that arise during the testing in the *Output Repository*.

在生成测试用例之后，Test Execution Environment 被激活，以便在同一个应用程序的许多实例上同时执行测试。为了可伸缩性，我们利用基于云的系统的并行性，在运行 Android Emulator 的虚拟节点上执行测试。除了代码覆盖之外，还实例化和部署了其他一些 android 特定的监视工具，如 Intent Sniffer [14]，以便在测试执行时收集运行时数据。这些监视工具记录在 Output Repository 中的测试期间出现的程序行为和错误(例如，崩溃、异常、访问冲突、资源抖动)。

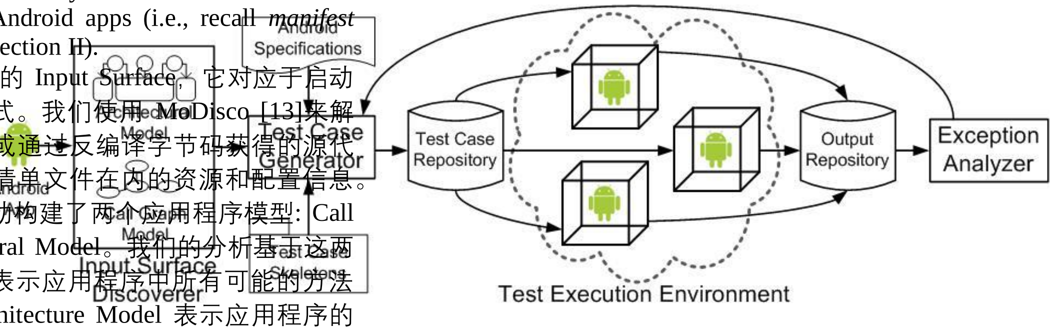


Figure 2. Overview of the approach. Components contained in the dotted bubble execute in parallel on the cloud.
图 2。方法概述。点状气泡中包含的组件在云中并行执行。

24

The *Exception Analyzer* engine then investigates the *Output Repository* to correlate the executed tests cases to the reported issues, and thus potential security vulnerabilities. Moreover, the *Exception Analyzer* engine prunes the collected data to filter any redundancy, since the same vulnerability may be encountered by multiple test cases. It also looks for anomalous behavior, such as performance degradations, which may also indicate vulnerabilities (e.g., an input that could instigate a denial of service attack).

Exception Analyzer 引擎然后调查 Output Repository，将执行的测试用例与报告的问题联系起来，从而发现潜在的安全漏洞。此外，Exception Analyzer 引擎会删除收集到的数据以过滤任何冗余，因为多个测试用例可能会遇到相同的漏洞。它还会查找异常行为，例如性能下降，这也可能表明存在漏洞(例如，可能引发分布式拒绝服务攻击的输入)。

V. Models of App
应用程序模型

As discussed in Section IV, our approach leverages two types of models for generating the test cases. Figure 3 depicts an example of the models that are automatically extracted for the Driving Direction app. The *Architectural Model*, partially depicted in Figure 3b, is generated by combining and correlating information containted in the configuration files and meta-data included in Android APK (i.e., *manifest* and layout XML files). Essentially, this model represents the app's architecture extracted from its configuration and resource files.

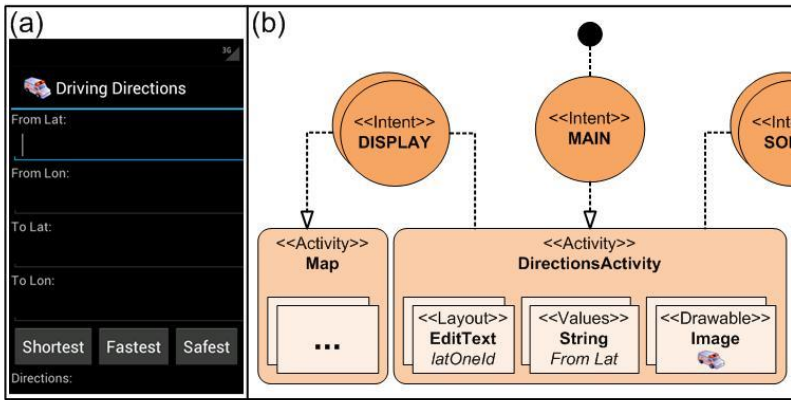
正如第四部分所讨论的，我们的方法利用了两种类型的模型来生成测试用例。图 3 描述了一个为 Driving Direction 应用程序自动提取模型的例子。架构模型(如图 3b 所示)是通过组合和关联包含在配置文件和包含在 Android APK 中的元数据(即清单和布局 XML 文件)中的信息而生成的。本质上，这个模型代表了从其配置和资源文件中提取出来的应用架构。

In Figure 3b, we use stereotypes to classify the different components types. *Activity* and *Service* are shown in orange rectangles, whereas *Intent* is shown in dark orange circles. Resources such as *Layout*, *Value*, and *Drawable* are shown in light orange rectangles. An example in each is shown using a triple tuple {*Classification*, *Type*, *Instance*}. For example, a *Layout* could be an *EditText* control with identifier *latOneId*. All of the resources used by an *Activity* or *Service* are contained within it. The *Intent* messaging between *Activities* and *Services* are represented by the dotted lines with the arrowhead showing the direction. For example, the Driving Directions Activity sends an *Intent* called *SOLVE* to the Route Solver Service.

在图 3b 中，我们使用原型对不同的组件类型进行分类。Activity 和 Service 显示在橙色矩形中，而 Intent 显示在深橙色圆圈中。诸如 Layout、 Value 和 Drawable 之类的资源以浅橙色矩形显示。每个示例都使用了三元组 { Classification , Type , Instance }。例如，一个 Layout 可以是一个带有 latOneId 标识符的 EditText 控件。一个 Activity 或者 Service 使用的所有资源都包含在其中。Activities 和 Services 之间的意图消息由虚线表示，箭头表示方向。例如，Driving Directions Activity 向 Route Solver Service 发送一个名为 SOLVE 的意图。

Initially, when the *Architectural Model* is built, the associations between *Layouts* and *Activities* are not known since they are set in code and are not present in the configuration files. To extract these associations, we update this model as described in the next section to associate the *Layouts* of an *Activity*. Moreover, we parse the app's source code using MoDisco [13] and extract all of the method invocation sequences. We use this information to derive the app's *Call Graph Model* as shown in Figure 3c. The *Call*

最初，在构建体系结构模型时，布局和活动之间的关联是未知的，因为它们是在代码中设置的，并且不存在于配置文件中。为了提取这些关联，我们更新了下一节描述的模型来关联一个活动的布局。此外，我们使用 MoDisco [13]解析应用程序的源代码，并提取所有的方法调用序列。我们使用这些信息来推导应用程序的 Call Graph 模型，如图 3c 所示。调用



Graph Model contains a set of call trees showing the different possible invocation sequences within a given application. Each yellow box in Figure 3c represents a method, and the lines represent the sequence of invocations. In *Tree 1*, the *DirectionsActivity*'s *onCreate()* method is called by the Android system, and it sets the layout of the *Activity* using a unique identifier reference. Then it finds each button using the button's identifiers and attaches a click listener for the button. *Tree 2* begins on the event of a button click. It calculates the appropriate driving directions and sets the output text. The link between *Tree 1* and *Tree 2* is implicit, and hence, shown as a dotted arrow. Initially this link is not present in the model; it is updated as described in the next section.

Graph Model 包含一组调用树，它们显示给定应用程序中不同的可能调用序列。图 3c 中的每个黄色方框代表一个方法，线条代表调用序列。在 Tree 1 中，DirectionsActivity 的 onCreate ()方法由 Android 系统调用，它使用唯一标识符引用设置 Activity 的布局。然后它使用按钮的标识符找到每个按钮，并为该按钮附加一个单击监听器。Tree 2 从一个按钮的点击事件开始。它计算出适当的驾驶方向并设置输出文本。Tree 1 和 Tree 2 之间的链接是隐式的，因此显示为虚线箭头。最初这个链接并没有出现在模型中; 它在下一节中被更新了。

Notice that the controls within the application are referenced across the models in Figure 3. For example, "*From Lat"* input box in Figure 3a is listed as a layout control with *latOneId* identifier in Figure 3b and accessed using this id in *Tree 2* of Figure 3c.

注意，应用程序中的控件在图 3 的模型中被引用。例如，图 3a 中的“ From Lat”输入框被列为一个布局控件，在图 3b 中使用 latOneId 标识符，并在图 3c 的 Tree 2 中使用该 id 访问。

VI. Discovering the Input Surface
发现输入曲面

In order to automatically perform input fuzzing of an app, we need to discover its input surface. We have developed a technique for automatically identifying the ways in which an Android app can be engaged as described next.

为了自动执行应用程序的输入模糊，我们需要发现它的输入面。我们已经开发了一种技术，可以自动识别 Android 应用程序的参与方式，如下文所述。

First, we identify the ways in which an app can be started. From the Android specification, we know that the main *Activity* launches when an app starts. We resolve this using our *Architectural Model*, which shows the *Intent* messages that each *Activity* is interested in receiving and responding. As seen in Figure 3b, in the case of Driving Direction app, the main *Activity* is *DirectionsActivity* that handles the *MAIN Intent*. Android specification also tells us that *onCreate()* method of the main *Activity* is the starting point in the *Call Graph Model* (see Tree 1 in Figure 3c).

首先，我们确定应用程序的启动方式。从 Android 规范中，我们知道当一个应用启动时，主活动就会启动。我们使用我们的建筑模型来解决这个问题，它显示了每个 Activity 感兴趣的接收和响应的 Intent 消息。如图 3b 所示，在 Driving Direction 应用的例子中，主要的活动是处理 MAIN Intent 的 DirectionsActivity。Android 规范还告诉我们，主活动的 onCreate ()方法是 Call Graph 模型的起点(参见图 3c 中的 Tree 1)。

Second, we need to determine how to navigate within the app to determine all the ways in which it can receive GUI inputs, as well as how it resumes, receives system notifications, and starts a *Service*. Unlike *Activity*, which only accepts GUI inputs, *Services* may receive inputs from other sources, which are also part of the input surface.

其次，我们需要确定如何在应用程序中导航，以确定它接收 GUI 输入的所有方式，以及它如何恢复、接收系统通知和启动服务。与只接受 GUI 输入的 Activity 不同，Services 可以接收来自其他来源的输入，这些来源也是输入界面的一部分。

To resolve these, we use the *Call Graph Model* described in the previous section. The root node of each tree is a

为了解决这些问题，我们使用上一节描述的 callgraph 模型。每个树的根节点是一个

Figure 3. Driving direction app: (a) screen shot of the app, (b) subset of the Architectural Model, and (c) subset of the Call Graph Model.
图 3。驾驶方向应用程序: (a)应用程序的屏幕截图，(b) architecture Model 的子集，以及(c) Call Graph Model 的子集。

25

method call that no other part of the application logic *explicitly* calls. Recall from Figure 1 that the lifecycle methods are called by the Android framework only. When these lifecycle methods are overridden in an app's implementation, they form the root nodes of that app's *Call Graph Model*. Similarly, the event methods of a *Service*, *onCreate()* and *onBind()* for example, would be root nodes. Some of these root nodes are the initiating points, where input may be supplied to the app from within or outside.
方法调用，应用程序逻辑的其他部分不显式调用。回想一下图 1，生命周期方法只被 Android 框架调用。当这些生命周期方法在应用程序的实现中被覆盖时，它们形成了应用程序的 Call Graph 模型的根节点。类似的，例如 Service 的事件方法 onCreate ()和 onBind ()就是根节点。这些根节点中的一些是起始点，其中的输入可以从内部或外部提供给应用程序。

Additionally, the controls on an *Activity* have handlers for their events. For example, a *Button* often has a click event associated with it. This event is handled by a class that implements the *OnClickListener* interface and overrides the *onClick()* method. We expect these sorts of handlers to be in the root nodes of our call trees as well, since Android is event driven and the event handlers are called by the Android system as opposed to the application logic. For instance, as depicted in Figure 3c, we see that *Tree 1*'s root is the *onCreate()* event handler, and *Tree 2*'s root node is the *onClick()* event handler.
另外，Activity 上的控件有事件的处理程序。例如，一个 Button 通常会有一个 click 事件与之相关联。这个事件由一个实现 OnClickListener 接口并覆盖 onClick () 方法的类来处理。我们希望这些类型的处理程序也位于调用树的根节点中，因为 Android 是事件驱动的，事件处理程序是由 Android 系统调用的，而不是应用程序逻辑。例如，如图 3c 所示，我们看到 Tree 1 的根是 onCreate ()事件处理程序，而 Tree 2 的根节点是 onClick ()事件处理程序。

From this point, we need to identify two additional attributes. We need to resolve what layout/view is being used by each *Activity*, since there can be many *Activities* with many layout XML files and their respective controls. Secondly, we also need to be able to link the implicit calls between the trees (recall the dotted arrow in Figure 3c).
从这一点出发，我们需要确定另外两个属性。我们需要解决每个活动使用的布局/视图，因为可能有许多带有许多布局 XML 文件及其各自控件的活动。其次，我们还需要能够链接树之间的隐式调用(回想一下图 3c 中的虚线箭头)。

In order to resolve this information, we traverse the call graph starting with the *onCreate()* root node of the main *Activity* and look for what layout is used. From the Android specification, we know this is achieved by calling the *setContentView()* method and passing it a reference identifier that describes the layout and controls. For instance, in the Driving Direction app, the view is set to the *Layout* with identifier *layout.main* (Figure 3c). Since this was set in the *DirectionsActivity*, we associate this *Activity* with this *Layout* in the *Architectural Model*. This means that the layout and controls identified in this way are those that get rendered on the screen when the corresponding *Activity* runs.
为了解析这个信息，我们从主 Activity 的 onCreate ()根节点开始遍历调用图，并查找使用的布局。从 Android 规范中，我们知道这是通过调用 setContentView ()方法并传递一个描述布局和控件的参考标识符来实现的。例如，在 Driving Direction 应用程序中，视图被设置为带有标识符 Layout.main 的 Layout（图 3c）。因为这是在 DirectionsActivity 中设置的，所以我们把这个 Activity 和这个 Layout 关联在建筑模型中。这意味着以这种方式标识的布局和控件是当相应的 Activity 运行时在屏幕上呈现的。

Finally, we continue down the graph and identify implicit method calls in order to link the different trees. We know that the links would have to be to other root nodes of trees,
最后，我们继续下图，并确定隐式方法调用，以便链接不同的树。我们知道这些链接必须指向树的其他根节点，

```
public class DirectionsTest extends
   ActivityInstrumentationTestCase2<DirectionsActivity>
公共类 directiontest 扩展了 ActivityInstrumentationTestCase2
{
 private Solo solo;
 私人独奏;

 public DirectionsTest() {
 公众方向测试
   super("edu.gmu.android", DirectionsActivity.class);}
   (" edu.gmu.android" , directionsactivityclass) ; }

 public void setUp() throws Exception {
 Public void setUp ()抛出 Exception {
   solo = new Solo(getInstrumentation(), getActivity());}
   Solo = new Solo (getininstrumentation () , getActivity
   ()) ; }

 public void testDirections() throws Exception {
 Public void testDirections ()抛出 Exception {
   solo.enterText(0, "38.95");
   solo.enterText (0, "38.95") ;
   solo.enterText(1, "77.46");
   solo.enterText (1, "77.46") ;
   solo.enterText(2, "38.95");
   solo.enterText (2, "38.95") ;
   solo.enterText(3, "77.46");
   solo.enterText (3, "77.46") ;
   solo.clickOnButton("Safest");  }
   单击按钮("最安全") ; }

 public void tearDown() throws Exception
   { solo.finishOpenedActivities(); }
 Public void tearDown ()抛出异常
   { solo.finishOpenedActivities ()}

}
```

Listing 1. Sample auto generated test case.
清单 1.Sample 自动生成的测试用例。

and achieved through setting event handlers. For example, system event handlers that handle notification events, such as when a call is received, network is disconnected, or the battery is running low. As trees are linked and connected, we traverse them in a similar fashion. By doing so, we are able to connect the entire call graph of the application, from beginning to end. Both the *Architectural Model* and the *Call Graph Model* are updated with the newly found information. Using this algorithm we can discover all the ways an app can be initiated, as well as the inputs it can receive.
并通过设置事件处理程序实现。例如，处理通知事件的系统事件处理程序，比如接到电话、断开网络或者电池电量不足。当树被链接和连接时，我们以类似的方式遍历它们。通过这样做，我们可以连接整个应用程序的调用图，从头到尾。架构模型和调用图模型都会根据新发现的信息进行更新。使用这个算法，我们可以发现一个应用程序可以启动的所有方式，以及它可以接收的输入。

VII. Test Case Generation
VII. 测试用例的生成

Now that the controls, their input value domain, the events, and their handlers have been inferred, the next step is to generate test cases for execution. We take a test case template, substitute our inferred information, and output the result as a Java file. Most of the generated text in the file is from the template, while the dynamic parts are replaced with the inferred information. For example, the class name is replaced with activity under test (e.g., *DirectionsActivity*), and the inputs are also generated as follows.

既然已经推断出了控件、它们的输入值域、事件和它们的处理程序，下一步是生成用于执行的测试用例。我们使用一个测试用例模板，替换我们推断的信息，并将结果输出为一个 Java 文件。文件中大部分生成的文本来自于模板，而动态部分被推断信息替换。例如，类名被替换为测试中的活动(例如，DirectionsActivity)，输入也生成如下。

For the test cases, our goal is to obtain sufficient code coverage, while generating inputs with security implications, e.g., an input that makes an application unavailable or violate access permissions. We attempt to generate our inputs so that we start with the first *Activity's* root node and traverse the tree, including any implicit connections, in a recursive manner. We monitor our code coverage by using *EMMA* [15], an open source toolkit that monitors and reports Java code coverage. By comparing the stack-trace reports generated by *EMMA* with the *Call Graph Model*, we can obtain an accurate assessment of code coverage.

对于测试用例，我们的目标是获得足够的代码覆盖率，同时生成具有安全隐患的输入，例如使应用程序不可用或违反访问权限的输入。我们尝试生成输入，以便从第一个 Activity 的根节点开始，并以递归方式遍历树，包括任何隐式连接。我们通过使用 EMMA [15]来监视我们的代码覆盖率，EMMA [15]是一个监视和报告 Java 代码覆盖率的开源工具包。通过比较 EMMA 生成的堆栈跟踪报告与 Call Graph 模型，我们可以获得代码覆盖率的准确评估。

We iteratively employ various *fuzzers* in order to generate and improve the inputs. The initial input generation for each *fuzzer* is based on using the Android specifications for each control and includes commonly employed rules, such as boundary values, very small or large values, special characters, empty values, etc. The valid input domain for an interface is derived by checking the specifications of a control. For example, we can tell whether the input domain of a text box is numerical, text, etc by referencing its specification. Based on the input domain, we employ the correct *fuzzer*. For instance, a *fuzzer* for numerical inputs starts off with negative numbers, zero, large values, and so on. In the case of text inputs, the text *fuzzer* generates null values, special characters, UTF characters, etc.

我们迭代地使用各种模糊器来生成和改进输入。每个模糊器的初始输入生成是基于对每个控件使用 Android 规范，包括常用的规则，如边界值、非常小或大的值、特殊字符、空值等。接口的有效输入域是通过检查控件的规范得到的。例如，我们可以通过引用文本框的规范来判断文本框的输入域是否是数字、文本等等。基于输入域，我们使用正确的模糊器。例如，用于数字输入的模糊器以负数、零、大值等开始。在文本输入的情况下，文本模糊器生成空值、特殊字符、 UTF 字符等。

We refine the inputs by using an iterative strategy, where we run test cases in iterations. We assess the depth of the call graph that a test case was able to penetrate. In the next iteration, we revise the inputs in one direction (lower/higher or negative/positive) and repeat the process. This way we are able to observe if we are obtaining coverage in the parts of the code that have not been tested.

我们通过使用迭代策略来细化输入，其中我们在迭代中运行测试用例。我们评估一个测试用例能够穿透的调用图的深度。在下一次迭代中，我们将输入修改为一个方向(低/高或负/正)，然后重复这个过程。通过这种方式，我们可以观察到我们是否获得了代码中未经测试的部分的覆盖率。

As part of our ongoing activity, we are developing more sophisticated input generation algorithms, which are further discussed in Section XI. But even with the current approach we have been very successful at generating a very large number of test cases, achieving substantial code coverage, and detecting real vulnerabilities.

作为我们正在进行的活动的一部分，我们正在开发更复杂的输入生成算法，这将在第十一节中进一步讨论。但是，即使使用当前的方法，我们也非常成功地生成了大量的测试用例，实现了大量的代码覆盖，并且检测到了真正的漏洞。

These test cases are stored inside of a test case repository database as shown in Figure 2. Currently, the generated test cases leverage the *Robotium* [16] framework. It is a testing framework built on Android's *JUnit* testing suite that facilitates automated testing.

这些测试用例存储在测试用例存储库数据库中，如图 2 所示。目前，生成的测试用例利用了 Robotium [16] 框架。它是一个基于 Android 的 JUnit 测试套件构建的测试框架，有助于自动化测试。

As an example, Listing 1 shows one of the many *Robotium* test cases automatically generated using our approach for the Driving Direction app. The name of the test case is *DirectionsTest* and it extends *ActivityInstrumentationTestCase2*, an *Android Activity* testing class supplied by the Android testing framework, which in turns extends from JUnit's *TestCase* class. JUnit's *setup()* and *tearDown()* methods are overridden to set up and finalize the *Activities* being tested. The test case has the *Robotium's Solo* object that is used to interact with the application, such as sending inputs. Essentially Solo mimics the human user of the app. Any method that starts with the word "*test*" is run when the test case executes. The generated *testDirections()* method uses the *Solo* instance to enter four numbers (two pairs of lat/long) in the input text boxes and then clicks on the *Safest* button. The input text is entered by using the index of the input field.

作为一个例子，清单 1 展示了使用我们的 Driving Direction 应用程序方法自动生成的许多 Robotium 测试用例之一。测试用例的名称是 directiontest，它扩展了 ActivityInstrumentationTestCase2，这是 Android 测试框架提供的一个 Android Activity 测试类，它又从 JUnit 的 TestCase 类扩展而来。JUnit 的 setup ()和 tearDown ()方法被重写以设置和完成被测活动。测试用例有 Robotium 的 Solo 对象，用于与应用程序交互，比如发送输入。本质上，Solo 模仿了应用程序的人类用户。任何以" test"开头的方法都会在测试用例执行时运行。生成的 testDirections ()方法使用 Solo 实例在输入文本框中输入四个数字(两对 lat/long)，然后单击最安全按钮。输入文本是通过输入字段的索引输入的。

In this test, note that both pairs of lat/long are set to the same value. In fact, this particular input along with the *Safest* combination exposed a bug in the routing algorithm: a divide by zero exception. The result of running this test case on the emulator is shown in Figure 4a. The integer divide by zero exception makes the application unavailable, thus making this test case a success for exposing a potential vulnerability.

在这个测试中，请注意两对 lat/long 被设置为相同的值。事实上，这个特殊的输入和最安全的组合暴露了路由算法的一个缺陷: 除以零的异常。在模拟器上运行这个测试用例的结果如图 4a 所示。整数除以零异常使应用程序不可用，因此这个测试用例成功地暴露了一个潜在的漏洞。

VIII. Test Execution Environment
测试执行环境

Fuzzing usually requires the execution of a large number of tests. This is challenging on both traditional desktops as well smartphones due to the limitation of resources and the length of time it takes to execute a large set of test cases. To mitigate this issue, we have developed a novel technique to execute the tests in *parallel* and on the cloud. This allows us to seamlessly scale up and down as necessary.

Fuzzing 通常需要执行大量的测试。由于资源的限制和执行大量测试用例所需的时间长度，这对于传统台式机和智能手机都是一个挑战。为了缓解这个问题，我们开发了一种新的技术，可以在云端并行执行测试。这使得我们可以根据需要无缝地进行扩展和缩小。

We have set up an instance of Amazon EC2 virtual server running Windows Server 2008, and configured it with Java SDK, Android SDK, Android Virtual Device, and a custom test execution manager engine developed by us. The execution manager is responsible for polling the test repository and running the test cases on its host environment. For each test, it launches the emulator, installs

我们已经建立了一个运行 Windows Server 2008 的 Amazon EC2 虚拟服务器实例，并用 Java SDK、Android SDK、 Android Virtual Device 和我们开发的自定义测试执行管理器引擎进行了配置。执行管理器负责轮询测试存储库并在其主机环境上运行测试用例。对于每个测试，它启动模拟器，安装

the app, and installs and executes the test. It is also responsible for persisting all results, along with log and monitor data to the output repository. We created a virtual machine *image* from our base machine configured in this way to be replicated on demand.

该应用程序，并安装和执行测试。它还负责保存所有的结果，以及日志和监控数据到输出存储库。我们从我们的基本机器上创建了一个虚拟机镜像，以这种方式进行配置，可以根据需要进行复制。

In order to execute the tests cases in parallel, we launch a set number of virtual node instances, built using our image template and using Amazon's EC2 API. The execution manager on each instance polls our test case repository database and executes the tests. As an example, a batch execution output of Listing 1 is shown in Figure 4b.

为了并行执行测试用例，我们启动了一组虚拟节点实例，它们使用映像模板和 Amazon 的 EC2 API 构建。每个实例的执行管理器调用我们的测试用例存储库数据库并执行测试。例如，清单 1 的批处理执行输出如图 4b 所示。

Using the above setup, we were able to run 1,000 test cases for the suite of apps shipped with EDS in less than *25 minutes* by using 100 parallel instances (already running) each processing 10 test cases. The same test cases took over *77 hours* to complete on a single workstation executing the tests sequentially. Note that the reported times are not just the execution time of tests, but also the time associated with loading the Android emulator, test setup, and clean up.

通过使用上面的设置，我们能够在不到 25 分钟的时间内，使用 100 个并行实例(已经在运行)，每个实例处理 10 个测试用例，为 EDS 附带的应用套件运行 1000 个测试用例。相同的测试用例需要 77 个小时才能在一个工作站上按顺序执行测试。请注意，报告的时间不仅仅是测试的执行时间，还包括与加载 Android 模拟器、测试设置和清理相关的时间。

IX. Test Case Results Analysis
测试用例结果分析

Currently, we categorize the observed/logged output information into *Interface*, *Interaction*, *Permissions*, and *Resources* types. *Interface* exceptions are caused by direct inputs at the surface, usually to GUI controls. *Interactions* exceptions are a result of communication failures with other components within the application or with external applications. *Permissions* exceptions are a result of access violations, such as the application attempting to access components or system APIs that it explicitly has not requested. *Resources* exceptions are caused by abnormal system usage causing resource depletion, unavailability, or certain operation to time out.

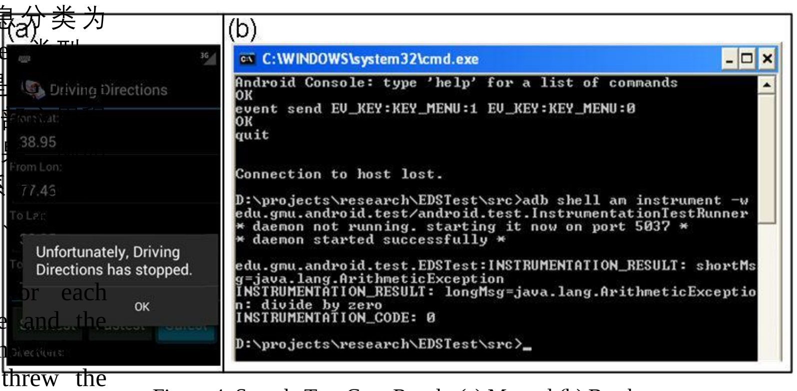
目前，我们将观察到的/记录的输出信息分类为 Interface、 Interaction、 Permissions 和 Resource
接口异常是由表面的直接输入引起的，通常是
件。交互异常是与应用程序内部其他组件或外
序通信失败的结果。权限异常是访问违规的结
应用程序试图访问它明确没有请求的组件或系
资源异常是由于异常的系统使用导致资源枯竭
或某些操作超时引起的。

Furthermore, we collect various attributes for each exception such as the exact Java exception type and the number of such exceptions, whether it was ch unchecked, and the Android system APIs that threw the exception. Example APIs are *Networking*, *Telephony*, *Internet*, *Media*, etc. We correlate exceptions information with the method, class, and package that contained the exception and the frequency with which the respective code block was exercised during the testing. This helps us identify and cluster the most defective components of the application, which could potentially aid with bug fix prioritization.

此外，我们还收集了每个异常的各种属性，比如确切的 Java 异常类型和这种异常的数量(无论它是否被选中)，以及抛出异常的 Android 系统 api。示例 api 是网络，电话，互联网，媒体等。我们将异常信息与包含异常的方法、类和包以及在测试期间执行相应代码块的频率相关联。这有助于我们识别和聚类应用程序中最有缺陷的组件，这可能有助于修复 bug 优先级。

We also analyze the code coverage by checking method invocations. A lack of depth in tree traversal or being unable to go beyond surface trees indicate that input is being filtered, application is branching early in different directions, or that only a certain input range is allowed. For example, in the Driving Direction app, we noticed that only a subset of test cases were able to penetrate deep into *Tree 2* in Figure 3c. This is because of input validation at the beginning of *Tree 2* that prevents the execution from going further unless valid latitude and longitude coordinates are entered. Since we knew all the nodes in *Tree 2*, we revised the inputs in the

我们还通过检查方法调用来分析代码覆盖率。树遍历的深度不够或者无法超越表面树表示输入正在过滤，应用程序正在向不同方向早期分支，或者只允许一定的输入范围。例如，在 Driving Direction 应用中，我们注意到只有一部分测试用例能够深入到图 3c 中的 Tree 2 中。这是因为 tree2 开头的输入验证阻止执行进一步，除非输入有效的经纬度坐标。因为我们知道 Tree 2 中的所有节点，所以我们修改了 Tree 2 中的输入


(a)
Driving Directions
From Lat:
38.95
From Lon:
77.45
To Lat:
Unfortunately, Driving Directions has stopped.
OK
(b)
C:\WINDOWS\system32\cmd.exe

```
Android Console: type 'help' for a list of commands
OK
event send EV_KEY:KEY_MENU:1 EV_KEY:KEY_MENU:0
OK
quit


Connection to host lost.

D:\projects\research\EDSTest\src>adb shell am instrument -w
edu.gmu.android.test/android.test.InstrumentationTestRunner
* daemon not running. starting it now on port 5037 *
* daemon started successfully *

edu.gmu.android.test.EDSTest:INSTRUMENTATION_RESULT: shortMs
g=java.lang.ArithmeticException
INSTRUMENTATION_RESULT: longMsg=java.lang.ArithmeticExceptio
n: divide by zero
INSTRUMENTATION_CODE: 0

D:\projects\research\EDSTest\src>_
```

Figure 4. Sample Test Case Result: (a) Manual (b) Batch
图 4。示例测试用例结果: (a)手册(b)批处理

27

subsequent iterations, and honed in on the inputs that could penetrate into Tree 2, and eventually all the way to the end of it. The iterative strategy helps us with obtaining greater code coverage, while at the same time be able to reason about valid input ranges for an app. This knowledge also helps us with finding inputs that are *outside* of the valid range, thus enabling negative test cases as well.
随后的迭代，并逐渐深入到可以渗透到树 2 中的输入，最终一直到树 2 的结束。迭代策略帮助我们获得更大的代码覆盖率，同时能够推理出一个应用程序的有效输入范围。这些知识也帮助我们找到超出有效范围的输入，从而使负面测试用例成为可能。

X. Related Work
相关工作

The Android development environment ships with a powerful testing framework [17] that is built on top of *JUnit*. *Robolectric* [18] is another framework that separates the test cases from the device or emulator and provides the ability to run them directly by referencing their library files. While these frameworks automate the execution of the tests, the test cases themselves still have to be written by the engineers.

Android 开发环境附带了一个强大的测试框架[17]，它构建在 JUnit 之上。Robolectric [18]是另一个将测试用例与设备或模拟器分开的框架，它提供了通过引用其库文件直接运行测试用例的能力。虽然这些框架自动执行测试，但测试用例本身仍然必须由工程师编写。

Traditionally *fuzz* testing tools use random inputs, but modern approaches utilize grammars for representing mutations of possible inputs [19][20] or achieve white-box *fuzz* testing using symbolic execution and dynamic test generation [21]. *SPIKE*, *Peach*, *File-Fuzz*, *Autodaf ́e* are examples of *fuzzers* that support some form of grammar representation. Applying exhaustive approaches are typically not feasible due to the path explosion problem.

传统的模糊测试工具使用随机输入，但现代的方法利用语法来表示可能输入的变化[19][20]，或使用符号执行和动态测试生成来实现白盒模糊测试[21]。SPIKE，Peach，File-Fuzz，Autodaf e 是支持某种形式的语法表示的模糊器的例子。由于路径爆炸问题，应用详尽的方法通常是不可行的。

Our research is related to the approaches described in [22][23] for testing Android apps. In [22], a crawling-based approach that leverages completely random inputs is proposed to generate unique test cases. [23] presents a random approach for generating GUI tests and uses the Android *Monkey* platform to execute them. We are leveraging reverse engineering techniques to obtain the app's implementation, and use program analysis to derive the test generation process. This sets us apart from these works that employ black-box testing techniques. Moreover, these approaches have neither targeted security issues, nor have they considered the scalability implications of their solutions.

我们的研究与[22][23]中描述的测试 Android 应用程序的方法有关。在[22]中，提出了一种基于爬行的方法，利用完全随机的输入来生成独特的测试用例。[23]提出了一种用于生成 GUI 测试的随机方法，并使用 Android Monkey 平台执行它们。我们正在利用逆向工程技术来获得应用程序的实现，并使用程序分析来推导测试生成过程。这使我们有别于那些使用黑盒测试技术的工作。此外，这些方法既没有针对安全问题，也没有考虑到他们的解决方案的可扩展性。

There has been a recent interest in using cloud to validate and verify software. TaaS is an automated testing framework that automates software testing as a service on the cloud
最近有人对使用云来验证和验证软件感兴趣。TaaS 是一个自动化测试框架，将软件测试自动化为云服务

[24]. *Cloud9* provides a cloud-based symbolic execution engine [25]. Similarly, our framework is leveraging the computation power of cloud to scale fuzz testing. Unlike prior research, however, by targetting our framework to Android, we are able to achieve significant automation.
Cloud9 提供了一个基于云的符号执行引擎[25]。类似的，我们的框架正在利用云计算的计算能力来扩展模糊测试。然而，与之前的研究不同的是，通过将我们的框架定位于Android，我们能够实现显著的自动化。

XI. Concluding Remarks
结束语

We have presented a novel framework for automated security testing of Android applications on the cloud. The key contributions of our work are (1) a fully automated test case generation, (2) iterative feedback loop to generate and guide our input in an intelligent manner that ensures code coverage and uncovers potential security defects, and (3) highly scalable *fuzzing* by leveraging the cloud.

我们提出了一个新的框架，用于在云上对 Android 应用程序进行自动安全测试。我们工作的主要贡献是: (1) 完全自动化的测试用例生成; (2)迭代反馈循环，以智能的方式生成和指导我们的输入，从而确保代码覆盖率和发现潜在的安全缺陷; (3)通过利用云实现高度可伸缩的模糊化。

In our on going work, we are exploring two approaches for improving the test case generation facet of our framework. First, we are developing an evolutionary algorithm for generating tests, as part of which we are modeling the problem of testing an Android app as a genetic
在我们正在进行的工作中，我们正在探索两种方法来改进我们框架的测试用例生成方面。首先，我们正在开发一种用于生成测试的进化算法，作为其中的一部分，我们正在将测试 Android 应用程序的问题建模为遗传问题

problem and developing an appropriate fitness function to evaluate the quality of test cases. Second, we are developing an Android-specific symbolic execution engine for automatically generating test cases. We are extending *Java Pathfinder*, which is capable of symbolically executing pure Java code, to work on Android. In addition, we are creating a graphical reporting environment that would allow the security analyst to visually explore the results of the testing, and in particular obtain metrics (e.g., achieved code coverage, bugs per KSLOC) that could then be used for making decisions as to the overall security and robustness.
问题和开发适当的适应度函数来评价测试用例的质量。其次，我们正在开发一个 android 特有的符号执行引擎，用于自动生成测试用例。我们正在扩展 Java Pathfinder，它能够象征性地执行纯 Java 代码，以适用于 Android。此外，我们正在创建一个图形化的报告环境，它将允许安全分析师可视化地探索测试结果，特别是获得度量标准(例如，实现的代码覆盖率、每个 KSLOC 的 bug)，这些度量标准随后可用于就总体安全性和健壮性做出决策。

ACKNOWLEDGMENTS
鸣谢

This research is supported by grant D11AP00282 from Defense Advanced Research Projects Agency.
这项研究得到了国防高级研究计划局 D11AP00282 的资助。

REFERENCES
参考文献

[1] A. Shabtai, et al., "Google Android: A comprehensive security assessment," *Security & Privacy, IEEE*, 8(2), pp. 35–44, 2010.
Shabtai 等，" Google Android: 一个全面的安全评估"，《安全与隐私》，IEEE，8(2)，第 35-44 页，2010。

[2] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House Publishers, 2008.
Takanen，j。DeMott，and c。Miller，Fuzzing for software security testing and quality assurance。Artech House Publishers，2008。

[3] C. Miller and C. Mulliner, "Fuzzing the Phone in your Phone," in *Black Hat Technical Security Conference USA*, 2009.
Miller 和 c. Mulliner，" Fuzzing the Phone in your Phone，" in Black Hat Technical Security Conference USA，2009。

[4] "Android Monkey." [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html.
在线版本: http://developer.Android. com/guide/developing/tools/Monkey. html。

[5] "Dalvik - Code and documentation from Android's VM team." [Online]. Available: http://code.google.com/p/dalvik/.
" Dalvik-来自 Android 虚拟机团队的代码和文档。"。

[6] "Android Developers Guide." [Online]. Available: http://developer.android.com/guide/topics/fundamentals.html.
在线版: http://developer.Android. com/Guide/topics/fundamentals. html。

[7] S. Malek, et al., "A style-aware architectural middleware for resource-constrained, distributed systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 256–272, 2005.
"资源受限、分布式系统的风格感知架构中间件"，软件工程，IEEE 事务，卷。第 31 期，第 3 期，第 256-272 页，2005 年。

[8] "Apktool." [Online]. Available: http://code.google.com/p/android-apktool/.
译自: http://code.google. com/。

[9] "Dex2jar." [Online]. Available: http://code.google.com/p/dex2jar/.
" Dex2jar. "[在线]可获得: http://code.google. com/p/Dex2jar/。

[10] "Smali." [Online]. Available: http://code.google.com/p/smali/.
" Smali。"[在线]可用于: http://code.google. com/p/Smali/。

[11] "Dedexer." [Online]. Available: http://dedexer.sourceforge.net/.
" Dedexer. "[在线]可获得: http://Dedexer.sourceforge. net/。

[12] "JD-GUI." [Online]. Available: http://java.decompiler.free.fr/?q=jdgui.
" JD-GUI。"[在线]可用: http://java.decompiler.free.fr/? q = jdgui。

[13] "MoDisco." [Online]. Available: http://www.eclipse.org/MoDisco/.
" MoDisco. "[在线]可访问: http://www.eclipse.org/MoDisco/。

[14] "Intent Sniffer." [Online]. Available: http://www.isecpartners.com/mobile-security-tools/intent-sniffer.html.
《意图嗅探器》(Intent Sniffer)，网址: http://www.isecpartners. com/mobile-security-tools/Intent-Sniffer. html。

[15] "EMMA." [Online]. Available: http://emma.sourceforge.net/.
《艾玛》(在线版)，可访问: http://EMMA.sourceforge. net/。

[16] "Robotium." [Online]. Available: http://code.google.com/p/robotium/.
网址: http://code.google.com/p/Robotium /。

[17] "Android Testing Framework." [Online]. Available: http://developer.android.com/guide/topics/testing/index.html.
" Android 测试框架。"[在线]可获得: http://developer.Android. com/guide/topics/Testing/index. html。

[18] "Robolectric." [Online]. Available: http://pivotal.github.com/robolectric/.
《机器电子》(Robolectric)[在线]可获得: http://pivotal.github. com/Robolectric/。

[19] K. Sen, D. Marinov, and G. Agha, *CUTE: A concolic unit testing engine for C*, vol. 30. ACM, 2005.
马里诺夫和 g。阿加，可爱: 一个结肠单元测试引擎 c，卷 30。ACM，2005 年。

[20] P. Godefroid, et al., "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Notices*, 2008, vol. 43, pp. 206–215.
Godefroid，et al。，"基于语法的白盒模糊"，in ACM SIGPLAN Notices，2008，vol. 43，P.. 206-215。

[21] P. Godefroid, et al., "Automated whitebox fuzz testing," *Network and Distributed System Security Symposium*, 2008, vol. 9.
网络与分布式系统安全研讨会，2008，第 9 卷，"自动化白盒模糊测试"。

[22] D. Amalfitano, et al., "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp. 252–261.
Amalfitano 等，"基于 GUI 爬行的 Android 移动应用程序测试技术"，在软件测试、验证及确认工作坊(ICSTW)，2011，第 252-261 页。

[23] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceeding of the 6th international workshop on Automation of software test*, 2011, pp. 77–83.
C. Hu 和 i. Neamtiu，" android 应用程序的 gui 自动化测试"，发表于 2011 年第 6 届软件测试自动化国际研讨会，第 77-83 页。

[24] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," *ACM symposium on Cloud computing*, 2010, pp. 155–160.
Candea，s. Bucur，and c. Zamfir，"自动化软件测试即服务"，ACM 云计算研讨会，2010，第 155-160 页。

[25] L. Ciortea, et al., "Cloud9: A software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.
Ciortea 等，" Cloud9: 一个软件测试服务"，ACM SIGOPS 操作系统评论，卷。43，no. 4，pp. 5-10,2010。