上下文感知的安卓应用程序漏洞检测研究

秦佳伟[1,2]，张华[1]，严寒冰[2]，何能强[2]，涂腾飞[1]
（1. 北京邮电大学网络与交换技术国家重点实验室，北京 100876；2. 国家计算机网络应急技术处理协调中心，北京 100029）

摘 要：针对基于学习的安卓应用程序的漏洞检测模型对源程序的特征提取结果欠缺语义信息，且提取的特征化结果包含与漏洞信息无关的噪声数据，导致漏洞检测模型的准确率下降的问题，提出了一种基于代码切片（CIS）的程序特征提取方法。该方法和抽象语法树（AST）特征方法相比可以更加精确地提取和漏洞存在直接关系的变量信息，避免引入过多噪声数据，同时可以体现漏洞的语义信息。利用 CIS，基于 Bi-LSTM 和注意力机制提出了一个上下文感知的安卓应用程序漏洞检测模型 VulDGArcher；针对安卓漏洞数据集不易获得的问题，构建了一个包含隐式 Intent 通信漏洞和 PendingIntent 权限绕过漏洞的 41 812 个代码片段的数据集，其中漏洞代码片段有 16 218 个。在这个数据集上，VulDGArcher 检测准确率可以达到 96%，高于基于 AST 特征和未进行处理的 APP 源码特征的深度学习漏洞检测模型。

关键词：安卓漏洞检测；深度学习；代码切片；漏洞语义特征
中图分类号：TP18
文献标识码：A
DOI: 10.11959/j.issn.1000−436x.2021198

Research on context-aware Android application vulnerability detection

QIN Jiawei[1,2], ZHANG Hua[1], YAN Hanbing[2], HE Nengqiang[2], TU Tengfei[1]
1. State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
2. The National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China

Abstract: The vulnerability detection model of Android application based on learning lacks semantic features. The extracted features contain noise data unrelated to vulnerabilities, which leads to the false positive of vulnerability detection model. A feature extraction method based on code information slice (CIS) was proposed. Compared with the abstract syntax tree (AST) feature method, the proposed method could extract the variable information directly related to vulnerabilities more accurately and avoid containing too much noise data. It contained semantic information of vulnerabilities. Based on CIS and BI-LSTM with attention mechanism, a context-aware Android application vulnerability detection model VulDGArcher was proposed. For the problem that the Android vulnerability data set was not easy to obtain, a data set containing 41 812 code fragments including the implicit Intent security vulnerability and the bypass PendingIntent permission audit vulnerability was built. There were 16 218 code fragments of vulnerability. On this data set, VulDGArcher's detection accuracy can reach 96%, which is higher than the deep learning vulnerability detection model based on AST features and APP source code features.

Keywords: Android vulnerability detection, deep learning, CIS, semantic characteristics of vulnerabilities

收稿日期：2021−08−10；**修回日期**：2021−09−27
通信作者：张华，zhanghua_288@bupt.edu.cn
基金项目：国家自然科学基金资助项目（No.62072051, No.61976024, No.61972048）；中央高校基本科研业务费专项资金资助项目（No.2019XD-A01）；教育部区块链核心计划基金资助项目（No.2020KJ010802）
Foundation Items: The National Natural Science Foundation of China (No.62072051, No.61976024, No.61972048), The Fundamental Research Funds for the Central Universities (No.2019XD-A01), Key Project Plan of Blockchain in Ministry of Education (No.2020KJ010802)

1 引言

近几年，安卓应用一直在快速增长中，但是随之增长的还有应用所产生的漏洞。2020 年，国家信息安全漏洞共享平台（CNVD, China National Vulnerability Database）收录安全漏洞中移动互联网漏洞占全年收录数量的 8.0%。因为所有的软件漏洞都存在被攻击者潜在利用的可能[1-3]，所以发现漏洞并修复它，才是避免软件遭受攻击的有效方法。2018 年，PIAnalyzer[4]分析并提取了 PendingIntent 权限绕过漏洞的规则，基于静态检测的方法实现对改漏洞的检测。目前，与 Intent 机制相关的研究[5-12]主要关注 APP 隐私泄露问题。过辰楷等[5]提出了一种基于安全要素语句插装的泄露检测方法。AmanDroid[7]通过跟踪 APP 组件间的交互信息来识别隐私泄露问题。

另外，为了降低人工依赖和提高对未知漏洞的发现能力，基于学习的漏洞检测成为技术发展趋势之一[13-25]。基于学习的漏洞检测研究主要集中在 Java 语言的漏洞方面。早期的基于学习的 Java 源代码漏洞检测研究[21,24]解决了基于规则的漏洞检测方法依赖人工经验的问题，但是漏洞代码的抽象表示缺乏语义特征，从而影响漏洞识别的准确率。2017 年，Ma 等[22]提出将 Java 代码转换成抽象语法树（AST, abstract syntax tree）的特征，然后采用机器学习模型来对 Java 程序进行漏洞检测。AST 特征可以保留程序对象之间的语义信息，但是包含与漏洞代码无关的噪声数据会导致误报[26-28]。Java 程序不具有 APP 的生命周期特性和组件间通信机制（ICC, inter-component communication），所以无法适用于检测无源码的 APP 的漏洞。2018 年，王持恒等[13]依据 APP 的流量数据和权限列表信息采用聚类方法检测广告漏洞，但是该方法仅适用于广告漏洞检测场景。2021 年，Gencer 等[29]研究直接依赖于时间的 APP 漏洞，并采用时间序列、多层感知器等多种模型生成了漏洞预警模型。

基于学习的 APP 漏洞检测缺少针对安卓本身运行机制所导致的漏洞的研究。现有的 2 种特征提取方法会导致漏洞检测的准确率下降。其中，基于代码中关键字符串计数的特征方法无法表现语义信息，也无法体现漏洞的上下文关联信息；AST 的漏洞提取方法会存在误报[26-28]。在针对其他编程语言的程序漏洞检测方面[15-16,19-20,30]，2019 年，Zou 等[15]针对 C 语言的程序漏洞提出一种名为 code gadgets 的程序特征表示方法，由此设计并实现了基于深度学习的漏洞检测系统。因为 C 语言的程序分析不涉及对回调方法的处理，所以该方法无法直接适用于 APP 漏洞检测。安卓 APP 不仅有 Java 语言的漏洞，还有不正确地使用平台的应用程序接口（API, application program interface）所导致的漏洞，危害更严重。例如，安卓 ICC 不仅允许同一个 APP 的不同组件间进行数据传递，而且允许不同的 APP 之间的数据传递。这就给 APP 带来了安全风险——使用该机制实现的功能的各个对象的属性设置都可能导致漏洞。因此不能通过 Java 代码的规则匹配检测相关漏洞。

要实现针对安卓本身运行机制的漏洞检测，且克服人工提取特征的局限性，需要解决以下 3 个问题。1) 目前缺少可供深度学习使用的 APP 漏洞数据集，如何获取一批可用的数据集？2) 面对安卓应用特有的 ICC 方式和无主程序入口的运行启动方式，如何进行程序分析和漏洞特征提取？3) APP 漏洞特征表示方法方面，如何在不缺少关键信息的情况下对程序进行语义化的特征抽象？

为了克服上述挑战，本文以隐式 Intent 通信漏洞（IISV, implicit Intent security vulnerability）和 PendingIntent 权限绕过漏洞（BPPAV, bypass PendingIntent permission audit vulnerability）为研究对象，针对安卓运行机制导致的漏洞检测提出了一种上下文感知的安卓应用程序漏洞检测方法。该方法可以从 APP 中提取出只与漏洞相关的代码信息，并且将特征代码中的自定义函数名与变量名进行统一格式化处理，既保留语义逻辑性也具有可读性。本文主要贡献如下。

1) 本文从 GooglePlay 获取了 5 000 个样本，采用工具和人工分析的方法对其进行漏洞标记，得到包含 IIS 的漏洞样本 1 806 个和包含 PLP 的漏洞样本 95 个，提取 41 812 条特征代码段。经特征化处理后，本文数据集与 APP 无直接关联关系，但是因为漏洞信息的敏感性，该数据集仅通过邮件提供。

2) 在 APP 漏洞特征表示上，本文提出一种包含语义信息的特征抽象方法——CIS。该方法可以保留程序的执行流程的结构信息，从 APP 中提取只与漏洞相关的代码信息，并且将特征代码中的自定义函数名与变量名进行统一格式化处理，既保留语义逻辑性也具有可读性。针对 APP 没有明确的唯一

主函数入口的情况，本文给出了 9 个入口点，可以更全面地构建 APP 内部代码逻辑与数据关系。

3) 基于 CIS 方法，本文选取 Bi-LSTM 算法构建了一个针对 APP 漏洞的深度学习检测模型 VulDGArcher，针对本文分析的 2 种漏洞，其识别准确率达到了 96%。

2　代码语义特征化

漏洞代码特征化应最大化保留语意信息和影响漏洞形成的因素。本文提出了一种特征抽象方法 CIS，利用 APP 漏洞点的上下文信息，提取与漏洞点有关的语义信息，减少无关变量和函数信息。

2.1　漏洞语义特征

APP 源码中包含很多逻辑处理流程，一个功能的实现需要用到其他的变量或者方法。在对某一种漏洞分析时，本文只关注和这个漏洞触发点相关的变量和方法，其他代码在这种场景下都是噪声。

以 IIS 为例，代码 1 是一个 APP（MD5 是 51da27661a8eff2f0cb37b7756e576b3 ）中使用隐式 Intent 的方法实现发送邮件的功能函数，第 11)行～第 15)行实现了一个隐式 Intent 对象，该对象中加入了过滤条件 Intent.ACTION_SENDTO，同时该对象中包含了全部邮件内容。该 APP 并没有设定发送邮件的 APP，也没有强制设置用户选择所有可以响应该 Intent 的应用程序，这种现象就会导致该 APP 存在 IIS。

代码 1　IIS 的示例代码

```
1) public void onBitcoinEmail(View arg0) {
2)  builder.setTitle(R.string.pwyl_send_to_ di-
alog_ title). setItems(toEmails.toArray(new String
[toEmails. size()]), new OnClickListener() {
3)        @Override
4)        public void onClick(DialogInterface di-
alog, int which) {
5)          String choice = toEmails.get(which);
6)          String email = "";
7)          if (!choice.equals(getString(R.string.
let_me_ select))) {
8)            email = choice;
9)          }
10)          /*set an intent action Intent. AC-
TION_SENDTO=安卓.intent. action. SENDTO*/
11)          Intent intent = new Intent (Intent.
ACTION_SENDTO);
12)          /*data settings*/
13)          intent.setData(Uri.parse("mailto:" +
email));
14)          intent.putExtra(Intent.EXTRA_ SUB-
JECT, subject);
15)          intent.putExtra(Intent.EXTRA_ TEXT,
body);
16)          /*Send an email by implict Intent */
17)          startActivity(intent);
18)        }
19)      });
20) }
```

综上，IIS 风险与 Intent 的使用有关。所以从 APP 的源代码角度分析，该风险应主要关注 Intent 的对象及其用到的方法。如代码 1 所示，影响该漏洞的代码只有第 12)行～第 18)行，其他是噪声数据。

代码 2 是某个 Android 系统中“设置”APP（CVE-2014-8609）使用 PendingIntent 方法实现添加用户功能。在第 3)行中，一个 PendingIntent 对象 mPendingIntent 被创建，并带有一个空的 Intent 对象。因为是系统内置 APP，所以 mPendingIntent 对象具有系统权限。当恶意 APP 注册接收该 mPendingIntent 对象时，因为 mPendingIntent 注册的 Intent 是空的，所以恶意 APP 可以修改 Intent 对象中的 Action 和 Extra data 属性，再以具有系统权限的 mPendingIntent 对象发送出去，以此达到权限绕过目的。

代码 2　PLP 的示例代码

```
1)private void addAccount(String accountType) {
2) Bundle addAccountOptions=new Bundle();
3)  mPendingIntent=PendingIntent.getBroadcast
(this, 0, new Intent(), 0);
4)  addAccountOptions.putParcelable (KEY_
CALLER_ IDENTITY, mPendingIntent);
5) addAccountOptions.putBoolean(EXTRA_ HAS_
MULTIPLE_USERS,Utils.hasMultipleUsers (this));
6)  AccountManager.get(this).addAccount( ac-
countType, null,
7) /* authTokenType */ null, /* requiredFeatures
*/ addAccountOptions,
```

8) null, mCallback, null /* handler */);

9) mAddAccountCalled = true;

10) }

综上分析，PLP 与 PendingIntent 和 Intent 的使用都有关，所以代码 2 中影响该漏洞的代码只有第 3)行和第 4)行。

特征化方法既要将漏洞相关代码表示出来，又要保留代码之间的语义信息的特征化方法。本文针对这一需求定义了一种特征化代码的方式 CIS，如式(1)所示。它是由全部影响漏洞存在的相关变量的上下文内容 C_i 组成的，也是由疑似漏洞点的直接或间接相关的所有代码组成的调用序列。

$$\mathrm{CIS}=\{C_k \mid i=1,\cdots,q\} \tag{1}$$

如式(2)所示，C_i 是一个有向图，由一个相关变量 i 的前向关系语句和后向关联的语句组成；v_i 是第 i 个变量的关系图中的某个调用语句。当同一个变量的数据从 v_{ik} 传递到 v_{iw} 时，这 2 个点之间存在边 e_{ikw}。

$$\begin{aligned} C_i &= \{(V'_i, E'_i) \mid < v_{ik}, v_{iw} > \in E'_i, \\ V'_i &= \{v_{ik} \mid k=1,\cdots,r\}\} \end{aligned} \tag{2}$$

代码 3 是代码 1 进行 CIS 处理后的结果。第 1)行表示声明了一个 Intent 对象且该对象的临时编号是 10；第 2)行表示对编号为 10 的这个 Intent 对象进行了初始化操作且初始化传递的参数是 String 类型；第 3)行～第 7)行依次表示编号为 10 的这个 Intent 对象进行设置属性等操作；第 8)行表示 Intent 对象被发送。

代码 3 IIS 的 CIS 示例代码

1) 10 = new <Intent>

2) 10.<init>(String;)

3) 36 = invokevirtual < StringBuffer, toString() String; > 11

4) 4 = invokestatic < CLASS1, FUN0(String;) String; > 2

5) 10.putExtra(String;String;)Intent; >13,4

6) 8 = new <String>

7) 10.putExtra(String;String;)Intent; >16,8

8) invokevirtual <Context, startActivity (Intent;) V > 1,10

以上结果显示 CIS 既包含 Intent 相关的代码也保证了语句的原本调用序列。CIS 在包含疑似漏洞点的所有相关对象和语句的同时，去掉了与它无关的代码信息，顺序性保留了逻辑上的语义信息。

2.2 构建 CIS

从一个 APP 文件构建关于一个疑似漏洞点的 CIS 的流程如图 1 所示。

1) 反编译

本文对 APP 进行漏洞分析的目标是 APK 文件，

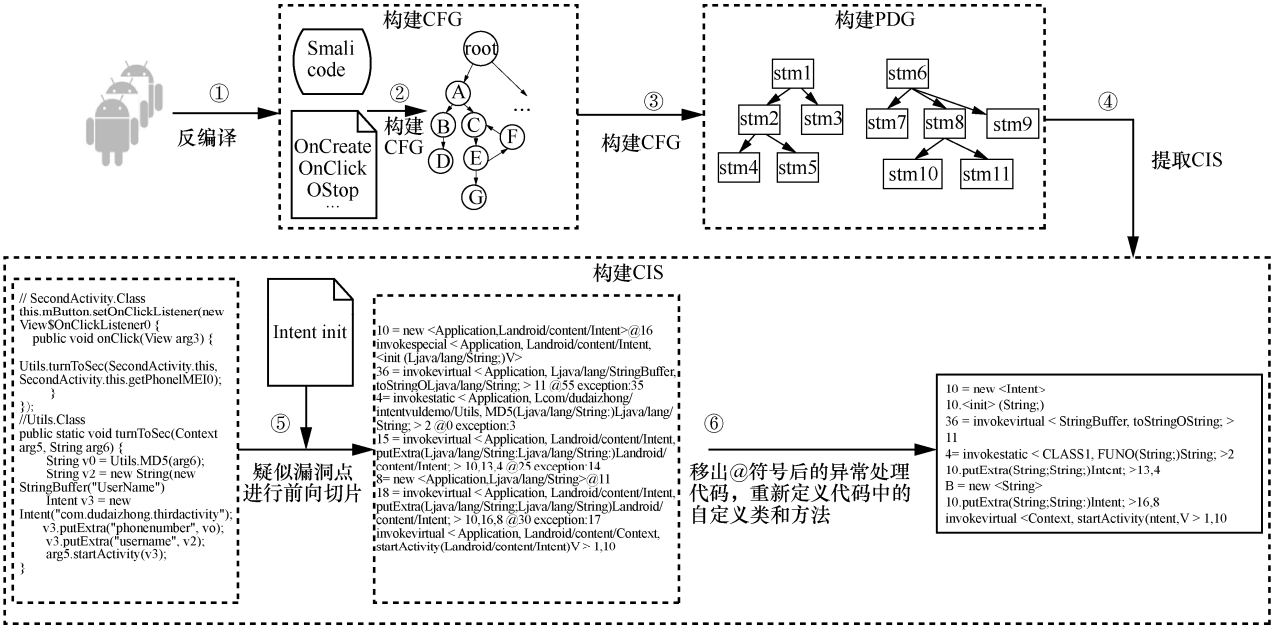


图 1 针对安卓应用程序提取 CIS 的框架

包含编译好的可执行文件。为了对源代码进行漏洞分析，本文基于 WALA 实现了对 APK 文件的反编译操作，从而获取到了 APP 的 Smali 形式的代码。

2) 构建控制流图

本文定义了如表 1 所示的入口点，并以此为基础为安卓 APP 构建必要的控制流图（CFG, control flow graph），如式(3)所示。

表 1　　定义的入口点

方法名	描述
onCreate (Bundle savedInstanceState)	初始化 activity 组件
onClick (View v)	用户点击操作调用
onStart ()	当用户将 activity 隐藏到后台调用
onCreate ()	初始化 service 组件
onStart (Intent intent)	开启 service 组件调用
onBind (Intent intent)	开始连接 service 组件
onUnbind (Intent intent)	停止与 service 组件连接
onRebind (Intent intent)	绑定服务时调用
onReceive (Context curContext, Intent broadcastMsg)	当接收来自其他 APP 的广播时调用

$$\text{CFG}=(N,\text{E},\text{nentry},\text{nexit})\quad |\ E=\{<n_k,n_w>|\ n_k,n_w\in N\} \tag{3}$$

其中，N 是全部节点集；程序中的每个语句都对应图中的一个节点 n_k，当 n_k 存在调用 n_w 的关系时，两者之前存在一条从 n_k 指向 n_w 的边 e_{kw}；nentry 和 nexit 分别为程序的入口和出口节点。经过分析安卓的 4 种组件的启动入口点和中间状态的转换关系以及安卓的 UI 反射入口等特性。

3) 构建程序依赖图

程序依赖图（PDG, program dependency graph）是程序的一种图形表示，是带有标记的有向多重图，如式(4)所示。构建方法是以程序的 CFG 为基础，去掉 CFG 的控制流边，加入数据和控制流边。因此 PDG 包括了数据依赖图和程序依赖图，数据依赖图定义了数据之间的约束关系，控制依赖图定义了语句执行情况的约束关系。PDG 全部节点集合为 V'，其中任意一个节点 s_k 表示语句或控制谓词表达式，边 E' 表示程序组成部分之间的依赖关系，包括控制依赖和数据依赖。如果 PDG 中的语句 s_k 和 s_w 可以通过控制流或者数据流来彼此关联，则两点之前存在一条边。因为 PDG 既包含程序中语句之间的数据依赖关系，又包含控制依赖关系，所以可减少漏洞信息搜索空间。

$$\text{PDG}=(V',E')\,|\,E'=\{<s_k,s_w>|\ s_k,s_w\in V'\} \tag{4}$$

4) 构建 CIS

本文按照算法 1 所描述的过程，选取一个疑似存在漏洞的风险点。如图 1 所示，IIS 漏洞的疑似漏洞点是 Intent < init >。本文首先提取疑似漏洞点的前向切片和后向切片；然后选取其中的配置方法，如 putExtra 等；最后对这类方法中的对象再次进行前向切片和后向切片提取。所有提取的中间切片结果使用 ConstructTree 存储在树结构中。因为本文最终得到的代码抽象结果要保持相对的逻辑顺序性，本文将使用 SearchTree 方法前序遍历读取树结构中的结果，并将所有结果汇总到一起形成这个疑似漏洞点的代码抽象形式 CIS'。

算法 1　基于疑似漏洞点构建代码抽象表示 CIS'

输入　存储切片代码 treeNode，一个疑似漏洞点 inp，APP 的程序依赖图 PDG

输出　疑似漏洞点的代码抽象表达式 CIS'

1) 初始化 CIS'= ∅

2) 取 inp 的前向切片 fslice= forwardSlice(PDG, inp)

3) if fslice ≠∅ then

4) 存储前向切片 ConstructTree(treeNode, fslice)

5) end if

6) 取 inp 的后向切片 bslice =backwardSlice (PDG, inp)

7) if bslice ≠ ∅ then

8) 存储后向切片 ConstructTree(treeNode, bslice)

9) end if

10) 在获取的切片代码中提取设置参数的方法 FSets =FindSettings(fslice)

11) 每一个涉及参数设定的方法都是新的疑似漏洞点 BSets=FindSettings(bslice)

12) for 循环遍历 前向切片结果 FSets 中的函数 do

13) 获取前向切片 fslice =forwardSlice(PDG, itemPoint)

14) if fslice ≠ ∅ then

15) 添加切片结果 ConstructTree(treeNode, fslice)

16) end if

17) 获取后向切片 bslice = backwardSlice (PDG, itemPoint)

18) if bslice ≠ ∅ then

19) 添加切片结果 ConstructTree(treeNode, bslice)

20) end if

21) end for

22) for 循环遍历后向切片结果 BSets 中的函数 itemPoint do

23) 获取前向切片 fslice =forwardSlice(PDG, itemPoint)

24) if fslice≠∅ then

25) ConstructTree(treeNode, fslice)

26) end if

27) 获取后向切片 bslice =backwardSlice(PDG, itemPoint)

28) if bslice≠ ∅ then

29) ConstructTree(treeNode, bslice)

30) end if

31) end for

32) 得到 treeNode 前序遍历 SearchTree (treeNode)结果为 CIS'

CIS'包含自定义的变量名和 API 等噪声数据，因此本节进一步优化数据特征以便模型更好地识别漏洞，优化过程如算法 2 所示。

算法 2 对 CIS'进行数据归一化处理

输入 初步提取的所有疑似漏洞点的抽象代码特征 CIS's

输出 最终的代码特征 CIS

1) for 循环遍历集合 CIS's 中的每个 CIS' do

2) 获取下标 Index =getIndex(CIS', '@')

3) 移出@后的异常信息 CIS'= stub(CIS', '@')

4) for 循环遍历 CIS'每一个自定义的变量值 var_i do

5) 将 var_i 替换成统一的命名 VAR_i

6) end for

7) for 循环遍历 CIS'每一个自定义的函数变量名 f_i

8) 将 f_i 替换成统一的命名 FUN_i

9) end for

10) for 循环遍历 CIS'每一个自定义的类变量名 c_i do

11) 将 c_i 替换成统一命名 $CLASS_i$

12) end for

13) for 循环遍历 CIS'每一个自定义的 API api_i

14) 简化 api_i 命名简化

15) end for

16) end for

17) 得到最后的漏洞特征 CIS

① 从 APP 提出来的 CIS'中包含了方法的异常处理信息，这些信息不会影响漏洞是否存在，但是会降低模型检测漏洞的准确率。经分析，这部分信息是在符号@后的字符。所以先获取@在 CIS'中的位置 Index，移除掉 Index 后面的表示异常处理的字符串，得到新的 CIS'。

② CIS'包含开发者自定义的变量信息。因为每个开发者自定义的变量命名不同，所以自定义变量就是噪声数据，会影响模型对漏洞的识别。本文将自定义方法进行统一的重新定义，计算出 CIS'中每个自定义的变量 var_i，然后按照先后顺序 i 对其进行替换变成统一的命名 VER_i，这样 CIS'中不同的自定义的变量只是编号不同，其他的都是用统一的 VER 字符串表示。利用同样的方法，依次将自定义的方法名变成 FUN_i，自定义的类名变成 $CLASS_i$。不同的类、方法和变量将按照后边的序号 i 进行区分，相同的类、方法和变量在不同位置命名保持一致。

③ 代码特征化的结果中，系统 API 的方法都是展示的完整的全限定位名，如 Java/lang/String，也就是包含了类所在的包名。因为 API 的类名就可以表现出该 API 的行为和意义，包名对于漏洞的识别是无意义的变量信息。尽管存在少量的 API 具有同样的类名，但是存在于不同的包名下，这些少量的 API 主要是监听类，并没有实际的行为意义，即不会影响漏洞的存在。所以计算出 CIS'中的每个以上类别的系统 API（api_i），去掉所有的包名只保留类名。例如，Java/lang/String 简化后为 String。

④ 对于上述几步的处理优化后，本文对同对象的调用方法进行再一次的组合，最大限度地还原 APP 的原本语句表达形式。最后得到含有最小噪声数据的代码特征值即 CIS。

3　基于语义代码片段的漏洞识别模型

本文基于 CIS 采用 Bi-LSTM 算法构建了一个保留语意信息的 APP 漏洞检测模型 VulDGArcher，图 2 为其训练过程。采用深度学习模型可以解决人工提取漏洞规则的漏报问题，并能克服依赖专家经验对新漏洞识别的局限性。

1) 构建 CIS

构建 CIS 的过程如算法 3 所示。

算法 3　构建 APP 的 CIS

输入　待分析的安卓应用 APP，所有的入口函数 EPs，疑似漏洞点集合 inps

输出　漏洞特征 CISs

1) 反编译得到 APP 的代码块 EntryBs =Decompile(APP, EPs)

2) for 循环遍历代码块集合 EntryBs 中的任意代码块 eb_i do

3)　　for 循环遍历代码块集合 EntryBs 中的其余任意代码块 eb_j do

4)　　if 代码块 eb_i 存在和 eb_j 的调用关系

5)　　这 2 个代码块之间存在边 e_i

6)　　end if

7)　　end for

8) end for

9) 按照上述流程构建 CFG

10) 计算所有的偏序关系 PO = <bb_i,bb_j>

11) 计算所有的依赖关系 CD = $bb_i \rightarrow bb_j$

12) for 循环遍历 CD 中的依赖关系 $bb_i \rightarrow bb_j$

13)　for 循环遍历块 bb_j 中的语句 v_{jk}

14)　构建一条从块 bb_i 指向 v_{jk} 控制边 e_{ijk} = $(\mathrm{br}(bb_i), v_{jk})$

15)　end for

16) end for

17) for 循环遍历数据依赖关系 $u \rightarrow v$ do

18)　if bb(u) < bb(v) then

19)　u 和 v 存在偏序关系，增加一条 $e_d = (u,v)$

20)　end if

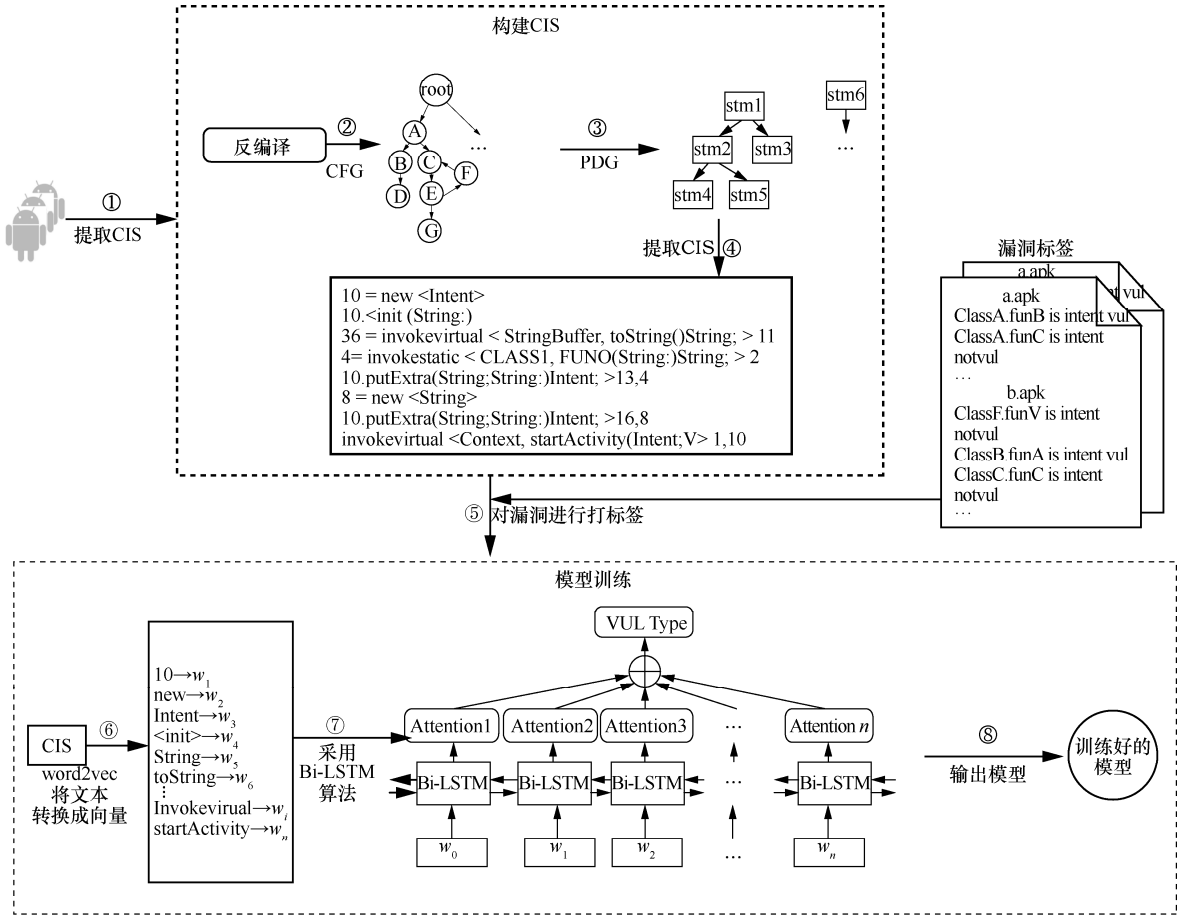


图 2　VulDGArcher 的模型训练过程

21) end for

22) 按照上述过程构建 PDG

23) 按照算法 1 构建一个疑似漏洞点 inp 的 CISs

① 本文基于 WALA 对 APP 逆向处理得到 smali 格式的代码。基于所有的入口点 EBs 处理得到 APP 的代码块集合 EntryBs。

② 从集合 EntryBs 中取出任意 2 个不同的代码块 eb_i 和 eb_j，如果从 eb_i 的最后一个语句到 eb_j 的前项存在条件分支或无条件分支，或者如果 eb_j 以程序顺序紧随 eb_i 且 eb_i 不以无条件分支结束，则在 eb_i 的基本块中添加一个有向边 e_{ij} 指向 eb_j。依次对所有的代码块进行同样的操作，就构建了 APP 的 CFG。

③ CFG 只能呈现 APP 的不同块之间的调用关系，为了分析具体 APP 内部的数据传递过程，还要进一步得到 APP 的数据传递情况。基于 CFG 计算所有偏序 PO 和所有的控制依赖关系 CD。针对 CD 中的每一个偏序关系 $bb_i \to bb_j$，对于 bb_j 中的每一个语句表达式 v_{jk}，都存在一条从 bb_i 指向 v_{jk} 的边 e_{ijk}。这样就会得到语句之间的控制关系。对于任意 2 个存在数据依赖关系的语句 u 与 v，如果其所在的偏序关系为 bb(u)<bb(v)，那么两者之间存在一条从 u 指向 v 的数据依赖边 e_{uv}。上述过程完成后，会得到由语句作为节点的 2 种关系图：一种用于表示控制依赖关系 CDG，另一种用于表示数据依赖关系 DDG。将 CDG 和 DDG 合并在一个图形中就生成了 PDG。

④ 基于 PDG，本文使用算法 1 对所有的疑似漏洞点构建对应的代码特征 CIS。这样得到的是关于每一个疑似漏洞点的包含语义的代码特征，其中还记录了这个疑似漏洞点所在的类名和方法名。

2) 数据集标注

CIS 的抽象特征没有包含它是否存在漏洞的标签，这一步将提取的代码特征进行漏洞打标签。

①本文将 APP 文件中的疑似漏洞点所在的函数使用 MobSF（mobile security framework）进行漏洞识别，对于识别出来的漏洞信息，本文再一次进行人工校验，最终整理出每个 APP 的漏洞信息。漏洞信息包含包名（p_j）、类名（c_j）、方法名（m_j）和漏洞类型（v_{jq}），如式(5)所示。

$$\mathrm{vIP}=\{\mathrm{vip}_j=(p_j,c_j,m_j,v_{jq})\mid j\in[1,n],q\in[0,2]\} \tag{5}$$

② CIS 中的一个疑似漏洞点 ip_j 如式(6)所示，其中 p'_j 表示包名，c'_j 表示类名，m'_j 表示方法名。

$$\mathrm{ip}_j=\{(p'_j,c'_j,m'_j)\mid j\in[1,n]\} \tag{6}$$

如式(7)所示，当 (p'_j,c'_j,m'_j) 等于 (p_j,c_j,m_j) 时，CIS 的漏洞标签 $\mathrm{CIS}_{\mathrm{tag}}$ 就是 v_j。

$$\mathrm{CIS}_{\mathrm{tag}}=\{v_j\mid(p'_j,c',m'_j)=(p_j,c,m_j),j\in[1,n]\} \tag{7}$$

3) 训练模型

经过上述阶段后的代码特征仍然是字符串的形式，这种格式的特征模型是无法直接识别的，所以也就无法将它当作输入变量。本文通过算法 4 将其转换成模型可接收的向量，具体过程如下。

算法 4 训练模型算法

输入 所有漏洞特征 CISs，模型定义的向量长度阈值 w

输出 训练好的模型 model

1) for 循环获取代码特征集 CISs 中的任一个特征 CIS do

2) 采用 word2vector 方法计算 CIS 的向量化 binData

3) if binData 长度小于 w then

4) binData 末尾补 0

5) end if

6) if binData 长度大于 w then

7) binData 进行截断

8) end if

9) binData 添加到训练集 trainData

10) end for

11) return 采用 Bi-LSTM 算法训练模型 model

① 本文使用 word2vector 对字符串形式的代码特征 CIS 进行向量化处理，得到模型可使用的词向量 binData。因为训练数据需要保证统一的长度，所以应对 binData 进行归一化处理。如果 binData 长度小于规定的阈值 w（本文 w=200），在 binData 后进行补 0 操作；如果 binData 长度大于阈值 w，从后边进行截断操作。最后统一存在训练数据集 trainData 中。

② APP 的漏洞代码特征是基于数据流和控制流构建的，其中包含了疑似漏洞点的上下文相关调用逻辑代码。深度学习算法应能够学习到漏洞代码块的调用逻辑。其次，APP 是否存在漏洞受疑似漏洞点的前向代码和后向代码的影响。因此选择的深

度学习网络应当满足如下特点：具有记忆性可以获取上下文关系；支持长句子，即长代码块；前向语句和后向语句的影响都能覆盖。技术成熟的深度学习网络中，Bi-LSTM 网络同时支持以上 3 个特性，可以作为 APP 漏洞检测的深度学习网络。

由于漏洞代码特征 CIS 都是较长的语句，为了学习长句子的语义信息，本文选取的 Bi-LSTM 模型中加入注意力机制，训练后得到漏洞识别模型。

4　实验和结果分析

本文用实验回答以下 3 个研究问题。

RQ1：CIS 能否识别出 APP 的多种漏洞？

RQ2：与现有的 APP 漏洞检测方法相比，VulDGArcher 的效果如何？

RQ3：VulDGArcher 的效率如何，是否具有实用性？

4.1　数据集和实验环境

因为相关的研究成果还没有开放出可用的安卓漏洞样本数据集，所以本文从 GooglePlay 获取了 5 000 个 APP 样本，先用安卓漏洞检测工具 MobSF 识别出 APP 的疑似漏洞点和初步的漏洞标记，再对检测后的 APP 进行人工的源码分析，分析应用程序中的疑似漏洞点上下文的数据流，标定出的所有 APP 中存在 IIS 或 PLP 的样本数量如表 2 所示。表 2 中的安全是指 APP 使用对应的 API 操作且安全的样本。因为存在同一个 APP 同时存在 IIS 和 PLP 这 2 种漏洞，所以在数据统计时没有去重。

表 2　数据集中存在不同的漏洞 APP 数量

漏洞名称	存在漏洞/个	安全/个	汇总/个
IIS	1 806	1 722	3 528
PLP	95	471	566
汇总	1 901	2 193	4 094

对 APP 进行 CIS 代码特征化处理后，现有 APP 样本中，属于某种漏洞的 CIS 特征条数如表 3 所示。为了对比实验，本文统计了现有 APP 中存在漏洞的原始代码文件（.class）的数量，如表 4 所示。因为同样一个源代码文件（.class）可能存在多个同类别的漏洞且表 4 中记录的是原始代码的文件数（.class 文件数），所以各种漏洞的数量和表 3 所展示的特征化后的结果不一样。

表 3　数据集中存在不同的漏洞 CIS 数量

漏洞名称	存在漏洞/个	安全/个	汇总/个
IIS	16 076	24 658	40 734
PLP	142	936	1 078
汇总	16 218	25 594	41 812

表 4　数据集中疑似漏洞点原始代码段数量

漏洞名称	存在漏洞/个	安全/个	汇总/个
IIS	11 633	17 169	28 802
PLP	133	876	1 009
汇总	11 766	18 045	29 811

4.2　评价指标

实验环境为一台 64 GB RAM， 3 TB SSD，Intel Intel Xeon CPU E5-2640 v2 2.00 GHz 服务器。

本文的实验是检测多个漏洞，所以是一个多分类问题。本文分别计算每个漏洞的对应的指标值，然后将全部类别的对应指标值进行取平均值。以第 i 类漏洞的检测为例，本文的评价指标如下。

真正类 TP_i：样本的真实类别是 i 漏洞，模型预测的结果也是 i 漏洞。

假负类 FN_i：样本的真实类别是 i 漏洞，模型预测的结果不是 i 漏洞。

假正类 FP_i：样本的真实类别不是 i 漏洞，模型预测的结果是 i 漏洞。

真负类 TN_i：样本的真实类别不是 i 漏洞，模型预测的结果不是 i 漏洞。

误报率（FPR, false positive rate）：不是 i 漏洞的样本被预测成 i 漏洞的占比。

$$\mathrm{FPR}=\sum_{i}^{n}\frac{\dfrac{\mathrm{FP}_i}{\mathrm{FP}_i+\mathrm{TN}_i}}{n} \tag{8}$$

召回率（TPR, true positive rate）：真实存在漏洞的样本判定为存在漏洞。

$$\mathrm{TPR}=\sum_{i}^{n}\frac{\dfrac{\mathrm{TP}_i}{\mathrm{TP}_i+\mathrm{FN}_i}}{n} \tag{9}$$

精确度（P, precision）：i 漏洞的样本被预测为漏洞的比例。

$$P=\sum_{i}^{n}\frac{\dfrac{\mathrm{FP}_i}{\mathrm{TP}_i+\mathrm{TN}_i}}{n} \tag{10}$$

F-Measure (F1)：精确度和召回率加权调和的平均。

$$\mathrm{F1}=\sum_{i}^{n}\frac{\dfrac{2P_i\mathrm{TPR}_i}{\mathrm{FP}_i+\mathrm{TN}_i}}{n} \tag{11}$$

精度（Acc, accuracy）：判为漏洞的样本数量占据样本总数量的比例。

$$\mathrm{Acc}=\sum_{i}^{n}\frac{\dfrac{\mathrm{TR}_i+\mathrm{TN}_i}{\mathrm{TP}_i+\mathrm{TN}_i+\mathrm{FP}_i+\mathrm{FN}_i}}{n} \tag{12}$$

4.3　实验分析

1) 回答 RQ1：CIS 特征在漏洞检测上的有效性

本文想验证基于 CIS 特征的深度学习检测模型是否能够正确地检测出漏洞，并且是否可以实现多种漏洞的检测。本文选取了 2 种安卓漏洞 IIS 与 PLP。在算法的选择上，由于 CIS 的特征包含了 interest point 的上下文，因此选取了具有后向反馈的 Bi-LSTM 作为模型的算法。

如表 5 所示，本文按照一定的比例将每种漏洞的数据拆分成了训练集、验证集和测试集。虽然其中 IIS 的数量大于 PLP 的数据量，但是在针对漏洞的 CIS 特征中包含了 interest point 的上下文语义信息，这样不同的漏洞之间的实现语句的明显不同，所以 CIS 会有明显的差别。这样不同漏洞之间的数据量的差距也不会影响模型识别的结果出现样本不均衡的假阳性问题。从图 3 中可以看出，CIS 特征在训练过程中随着训练迭代次数的增加，训练集和验证集的 auc 参数都在逐步增加并逐渐达到一定的平稳状态，采用时训练集和验证集的 loss 参数都在逐步的下降并达到平稳态。这说明基于 CIS 的特征是可以实现针对安卓多种漏洞检测。

表 5　用于模型训练和验证的数据集的分布

数据集	IIS			PLP		
	存在漏洞/个	安全/个	汇总/个	存在漏洞/个	安全/个	汇总/个
训练集	11 253	8 630	19 883	100	180	280
验证集	1 607	960	2 567	14	20	34
测试集	3 216	2 400	5 616	28	50	78
汇总	16 076	11 990	28 066	142	250	392

为了进一步验证基于 CIS 特征的模型的其他指标项随着训练样本数量的不同的变化情况，本文采用不同的训练集进行模型训练，训练后的模型都采用同样的测试数据集进行测评模型的各种指标项。在保证测试集不变的情况下，本文将表 5 中的训练集分别拆分成 8 组，详细如表 6 所示。本文针对每组训练集都采用同样的模型和参数配置，图 4 显示的是不同的训练集下的 CIS 的特征的模型的漏洞检测指标变化。从图 4 可以看出，随着训练样本集的扩增，模型的精确度、召回率、F-Measure 和精度都在逐步增长并达到最好的 98%，误报率逐步下降到 2%。

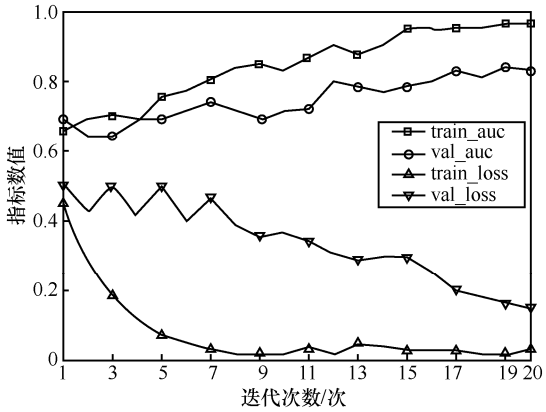


图 3　VulDGArcher 在训练集和测试集中的不同指标变化曲线

表 6　用于模型训练和验证的数据集的分布

组	IIS/个	PLP/个	汇总/个
1	2 200	34	2 234
2	4 400	68	4 468
3	6600	102	6 702
4	8 800	136	8 936
5	11 000	170	11 170
6	13 200	204	13 404
7	15 400	238	15 638
8	17 015	275	17 290

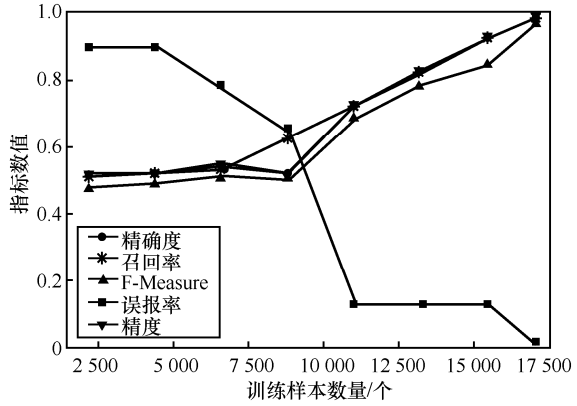


图 4　VulDGArcher 在不同的训练集数量下的各种指标变化

基于 CIS 特征的深度学习检测模型 VulDGArcher 可以正确地检测出 APP 中的多种漏洞。Bi-LSTM 算法可以捕获长句子的双向语义信息，基于该算法的漏洞检测模型精确度可以达到 98%，所以 Bi-LSTM 算法更适用于 VulDGArcher 的检测任务。

2) 回答 RQ2：VulDGArcher 与不同的漏洞检测方法进行对比漏洞检测效果

为了验证 VulDGArcher 中 CIS 的漏洞特征方法的有效性，本文采用不同的模型算法和不同的特征提取方法进行交叉实验。本文在模型算法方面选取 Bi-LSTM 和 CNN；在 APP 的漏洞特征表示方法方面选取 APP 的 Java 源代码文本特征[24]、APP 的 AST 特征[22]和 CIS。APP 的 Java 源代码文本特征（CB, code block），就是对待检测的目标 APP 进行逆向处理，获取其源代码信息，去掉这些源代码信息中无意义的标记符号，最后转换成深度学习模型可识别的向量。代码 4 是一个 APP 的 IIS 漏洞原始代码样例。APP 的 AST 特征就是将 APP 逆向获取出来的源代码信息进行 AST 表示，去掉其中的特殊标记符，然后将其转换成深度学习可识别的向量信息。图 5 是代码 4 所示的漏洞方法的 AST 表示，由于篇幅限制，图 5 中只展示了与 Intent 相关的 AST。

代码 4　IIS 的示例代码

```
1) public class AboutDialogFragment extends DialogFragment {
2)  @Override
3)  public Dialog onCreateDialog(Bundle savedInstanceState) {
4)      View view = li.inflate(R.layout.about, null);
5)      TextView authors = (TextView) view.findViewById (R.id.authors);
6)      TextView[] links = { authors,
7)          (TextView) view.findViewById (R.id.lib_list),
8)          (TextView) view.findViewById (R.id. github_issues) };
9)      for (TextView link : links)
10)         ((TextView) view.findViewById(R.id.btc_addr)).setOnClickListener(new OnClickListener() {
11)         @Override
12)         public void onClick(View v) {
13)             String btcLink = currentActivity.getString (R.string.btc_link);
```

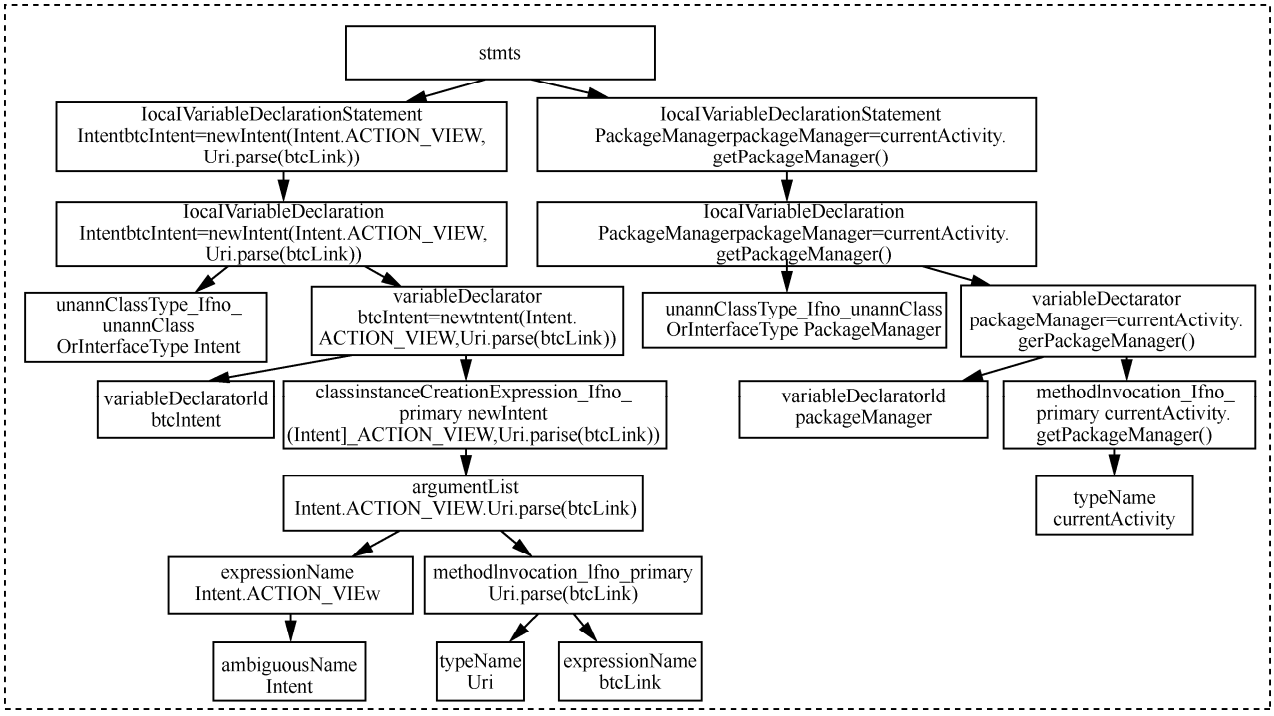


图 5　抽象语法树表示形式

```
14)  /*send an url to an APP that can parse url and */
15)  Intent btcIntent = new Intent (Intent.ACTION_VIEW, Uri.parse(btcLink));
16)  PackageManager packageManager = currentActivity.getPackageManager();
17)  /*Get all APPs that can respond to the Intent*/
18)  List<ResolveInfo> activities = packageManager.queryIntentActivities(btcIntent, 0);
19)  boolean isIntentSafe = activities.size() > 0;
20)  if(isIntentSafe) {
21)  /*Open the activity of one APP*/
22)  startActivity(btcIntent);}}});
23)  return builder.create();}
24) }
```

表 7 是不同模型在相同测试集上的结果。从表 7 可以看出，基于原始的代码做漏洞检测模型的误报比较高，达到 51%的原因包括：1) 原始的代码中包含了大量和漏洞形成无关的噪声数据；2) 原始代码中的自定义方法在每个 APP 中都是不一样的。相比较，VulDGArcher 在代码特征化的过程中提取的是疑似漏洞点的语义上下文信息，去除了所有无关的代码，并且对于 APP 自定义的方法名和类名也进行了归一化处理，这样才能使 VulDGArcher 的误报率和漏报率很低。图 6 可以更加直观地反映 VulDGArcher 与基于其他 2 种特征化方法的模型的效果。由图 6 可知，VulDGArcher 的 APP 漏洞检测效果更好。CNN 的漏洞检测模型虽然也可以识别出漏洞，但是模型测试的各个指标都低于 Bi-LSTM 算法。图 7 是同样的数据下 2 种算法的 ROC 曲线。从图 7 可以直观地看出，Bi-LSTM 算法更适合 CIS 特征。

表 7 不同类型的代码特征的数据进行模型训练的效果

模型	F1	FPR	P	TPR	Acc
code block + Bi-LSTM	0.71	0.51	0.74	0.69	0.69
AST + Bi-LSTM	0.84	0.12	0.84	0.84	0.84
CIS + Bi-LSTM (VulGDArcher)	0.98	0.02	0.98	0.98	0.98
code block + CNN	0.70	0.52	0.71	0.71	0.70
AST + CNN	0.85	0.13	0.83	0.83	0.83
CIS + CNN	0.91	0.13	0.92	0.88	0.89

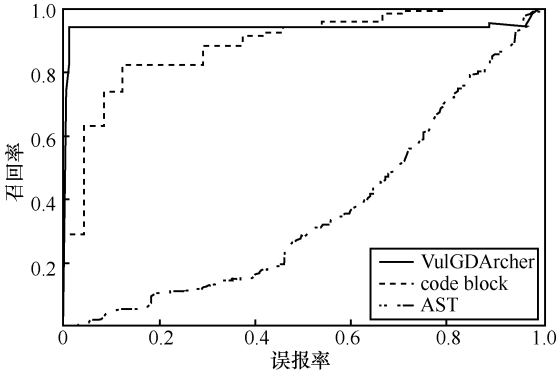


图 6 基于不同的漏洞特征方法的漏洞检测模型的 ROC 曲线

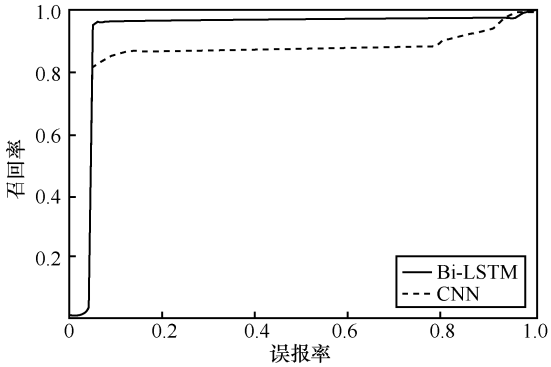


图 7 同样的数据条件下，2 种不同算法的 ROC 曲线

为了验证上述观点，本文提取一个存在 IIS 风险的 APP，在实验验证过程中，VulDGArcher 可以识别出它存在该风险，但是基于原始代码文件的模型无法正确判断该 APP 的这个风险。代码 5 是代码 4 经过 VulDGArcher 将原始代码进行语义特征处理化后的结果。从代码 5 中可以看出，特征化后的代码只包含和 IIS 风险相关的语法信息。

代码 5 VulDGArcher 对 IIS 提取特征的结果

```
1) 8 = new <Intent>
2) 8.invokespecial <Intent, <init>(String; Uri;)V >9,11
3) 17 = invokevirtual < PackageManager, queryIntentActivities(Intent;I)List; > 15,8,3
4) invokevirtual < CLASS1, startActivity(Intent;)V > 22,8
```

相较于原始代码，代码 5 去除了很多无用的噪声数据。这也反映了 VulDGArcher 在漏洞识别上具有很好的效果，主要取决于基于语义的代码特征化处理。

通过上述结果可以看出，基于 CIS 漏洞特征化的深度学习检测模型在 APP 漏洞检测上效果更优。因为 CIS 相较其他 2 种方法提取的漏洞特征去掉了

无用的代码信息，且包含的信息是疑似漏洞点的上下文相关的代码信息。

3) 回答 RQ3：VulDGArcher 与不同的漏洞检测方法验证实用性

本文通过实验从检测漏洞的效率角度评估 VulDGArcher 是否具有实用性。本实验增加 2 种开源的基于规则的安卓漏洞检测器：AndroBugs 和 Marvin-static-Analyzer。它们在 Github 上具有很多的 forks and stars，都支持 IIS 脆弱性检测，但是不支持 PLP 漏洞检测。本文将基于 PIAnalyzer[4]分析出的 PLP 检测规则配置在 AndroBugs 和 Marvin-static-Analyzer 工具中，使工具支持对 2 个漏洞的检测。本文从测试的数据集中随机地选取了 100 个 APP，然后分别采用 AndroBugs、Marvin-static-Analyzer、基于 AST 的深度学习漏洞检测模型和 VulDGArcher 进行漏洞检测。如表 8 所示，针对同样的漏洞，基于学习的漏洞检测方法的准确率高于基于规则的检测方法。因为这 2 个漏洞的检测方法需要考虑到漏洞点的上下文环境中涉及的多种对象属性配置，而基于规则的检测方法无法覆盖上述条件，所以检测准确率相对较低。

表 8　　不同检测方法的准确率

漏洞	IIS	PLP
Marvin-static-Analyzer	0.42	0.72
AndroBugs	0.53	0.85
AST	0.84	0.83
VulDGArcher	0.98	0.96

图 8 是不同工具的耗时情况。VulDGArcher 的耗时主要是代码特征化处理，模型的漏洞识别部分接近毫秒级。因为 VulDGArcher 需要构建 APP 的 PDG，然后对提取出来的特征进行向量化处理，所以这部分处理是耗时的过程。虽然如此，VulDGArcher 平均在 5 s 以内就可以完成对一个 APP 的漏洞检测。图 9 和图 10 分别是各漏洞检测工具的 CPU 消耗比例和内存消耗比例。从图 9 和图 10 可以看出，VulDGArcher 的内存消耗最多为 22%，远低于 Marvin-static-Analyzer 的 48%; CPU 资源使用最多为 30%，并且随着运行逐渐保持低于 10%。

同种运行环境下，本文将 VulDGArcher 与其他 3 种漏洞检测引擎进行比较，从准确率、耗时和资源消耗等方面衡量发现，VulDGArcher 的代码特征化是相对耗时的，但是并不是极其耗费资源的。所以本文建议通过并发运行平衡掉耗时的缺点。

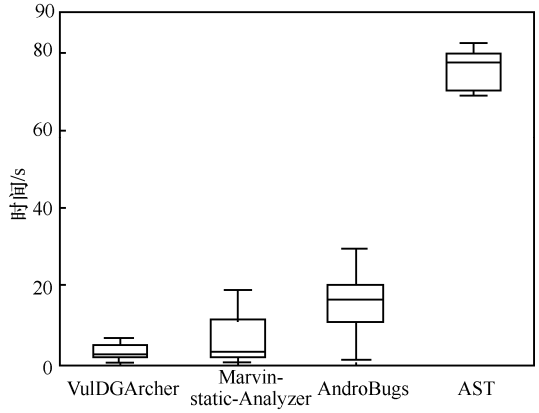


图 8　不同种检测方法的时间消耗

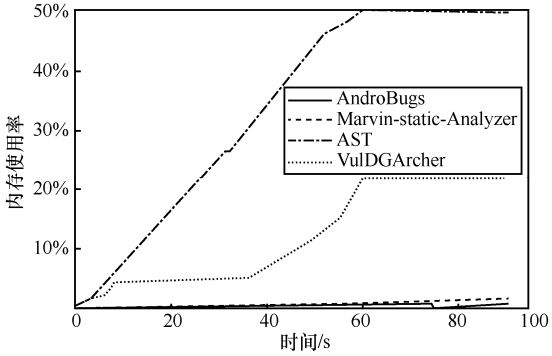


图 9　不同种检测方法的内存消耗

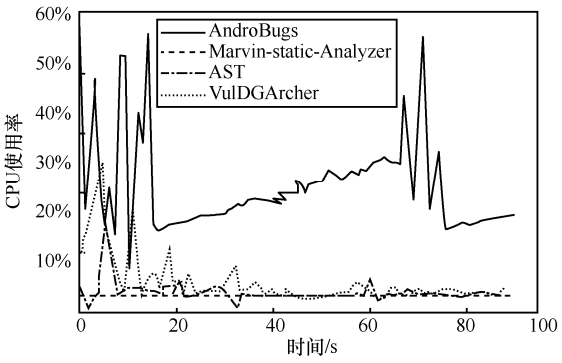


图 10　不同种检测方法的 CPU 消耗

5　结束语

本文针对现有基于学习的 APP 漏洞检测方法的特征表示结果缺乏语义信息而影响漏洞检测准确率的问题，提出了一种包含上下文语义信息的特征抽象方法 CIS，并对由 APP 的 API 误用导致的漏洞提出了一种上下文感知的检测方法。CIS 可以从 APP 中提取只和漏洞相关的变量信息，并且可以消除开发者自定义代码的类名、方法名和变量名对模型检测效果的影响。本文基于 CIS 采用 Bi-LSTM 和

注意力机制构建了一个 APP 漏洞检测模型 VulDGArcher。与基于 AST 和原始代码为特征化的检测方法相比，VulDGArcher 可以有效识别 APP 不正确使用安卓平台的 API 所导致的漏洞。此外，本文构建了一个包含 41 812 条特征代码段的数据集。

参考文献：

[1] CHOWDHURY I, ZULKERNINE M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities[J]. Journal of Systems Architecture, 2011, 57(3): 294-313.

[2] YAMAGUCHI F, WRESSNEGGER C, GASCON H, et al. Chucky: exposing missing checks in source code for vulnerability discovery[C]//Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. New York: ACM Press, 2013: 499-510.

[3] 赵尚儒, 李学俊, 方越, 等. 安全漏洞自动利用综述[J]. 计算机研究与发展, 2019, 56(10): 73-87.
ZHANG S R, LI X J, FANG Y et al. An overview of automatic exploitation of security vulnerabilities[J]. Computer Research and Development, 2019, 56(10): 73-87.

[4] GRO S, TIWARI A, HAMMER C. PIAnalyzer: a precise approach for PendingIntent vulnerability analysis[C]//Computer Security. Berlin: Springer, 2018: 41-59.

[5] 过辰楷, 许静, 司冠南, 等. 面向移动应用软件信息泄露的模型检测研究[J]. 计算机学报, 2016, 39(11): 2324-2343.
GUO C K, XU J, SI G N, et al. Model checking for software information leakage in mobile application[J]. Chinese Journal of Computers, 2016, 39(11): 2324-2343.

[6] WEI F G, ROY S, OU X M, et al. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps[C]//Proceedings of the ACM Conference on Computer and Communications Security. New York: ACM Press, 2014: 1329-1341.

[7] KLIEBER W, FLYNN L, BHOSALE A, et al. Android taint flow analysis for app sets[C]//Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. New York: ACM Press, 2014: 1-6.

[8] BAGHERI H, SADEGHI A, GARCIA J, et al. COVERT: compositional analysis of android inter-app permission leakage[J]. IEEE Transactions on Software Engineering, 2015, 41(9): 866-886.

[9] LI L, BARTEL A, BISSYANDÉ T F, et al. IccTA: detecting inter-component privacy leaks in android apps[C]//Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Piscataway: IEEE Press, 2015: 280-291.

[10] OCTEAU D, MCDANIEL P, JHA S, et al. Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis[C]//Proceedings of the 22nd USENIX Conference on Security. Berkeley: USENIX Association, 2013: 543-558.

[11] OCTEAU D, LUCHAUP D, DERING M, et al. Composite constant propagation: application to android inter-component communication analysis[C]//Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Piscataway: IEEE Press, 2015: 77-88.

[12] LEE Y K, BANG J Y, SAFI G, et al. A SEALANT for inter-app security holes in android[C]//Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2017: 312-323.

[13] 王持恒, 陈晶, 苏涵, 等. 基于宿主权限的移动广告漏洞攻击技术[J]. 软件学报, 2018, 29(5): 1392-1409.
WANG C H, CHEN J, SU H, et al. Mobile advertising loophole attack technology based on host APP's permissions[J]. Journal of Software, 2018, 29(5): 1392-1409.

[14] DAM H K, TRAN T, PHAM T, et al. Automatic feature learning for predicting vulnerable software components[J]. IEEE Transactions on Software Engineering, 2021, 47(1): 67-85.

[15] ZOU D Q, WANG S J, XU S H, et al. μVulDeePecker: a deep learning-based system for multiclass vulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.

[16] PERL H, DECHAND S, SMITH M, et al. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2015: 426-437.

[17] SCANDARIATO R, WALDEN J, HOVSEPYAN A, et al. Predicting vulnerable software components via text mining[J]. IEEE Transactions on Software Engineering, 2014, 40(10): 993-1006.

[18] BAN X B, LIU S G, CHEN C, et al. A performance evaluation of deep-learnt features for software vulnerability detection[J]. Concurrency and Computation: Practice and Experience, 2019, 31(19): e5103.

[19] LIN G J, ZHANG J, LUO W, et al. Cross-project transfer representation learning for vulnerable function discovery[J]. IEEE Transactions on Industrial Informatics, 2018, 14(7): 3289-3297.

[20] WU F, WANG J G, LIU J Q, et al. Vulnerability detection with deep learning[C]//Proceedings of 2017 3rd IEEE International Conference on Computer and Communications. Piscataway: IEEE Press, 2017: 1298-1302.

[21] HOVSEPYAN A, SCANDARIATO R, JOOSEN W, et al. Software vulnerability prediction using text analysis techniques[C]//Proceedings of the 4th International Workshop on Security Measurements and Metrics. [S.l.:s.n.], 2012: 7-10.

[22] MA S Q, THUNG F, LO D, et al. VuRLE: automatic vulnerability detection and repair by learning from examples[C]//Computer Security – ESORICS 2017. Berlin: Springer, 2017: 229-246.

[23] 乐洪舟, 张玉清. 网络直播平台主播地理位置泄露漏洞的分析与利用[J]. 计算机学报, 2019, 42(5): 1095-1111.
YUE H Z, ZHANG Y Q. Vulnerability analysis and exploitation of location privacy leakage in webcasting platforms[J]. Chinese Journal of Computers, 2019, 42(5): 1095-1111.

[24] AVERSANO L, CERULO L, DEL GROSSO C. Learning from bug-introducing changes to prevent fault prone code[C]//Proceedings of Ninth International Workshop on Principles of Software Evolution in

Conjunction with the 6th ESEC/FSE Joint Meeting. [S.l.:s.n.], 2007: 19-26.

[25] GARG S, BALIYAN N. A novel parallel classifier scheme for vulnerability detection in Android[J]. Computers & Electrical Engineering, 2019, 77: 12-26.

[26] CURTSINGER C, LIVSHITS B, ZORN B, et al. ZOZZLE: fast and precise in-browser JavaScript malware detection[C]//Proceedings of the 20th USENIX Conference on Security. Berkeley: USENIX Association, 2011: 3.

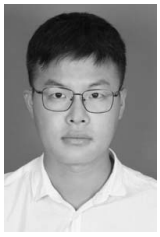
[27] RIECK K, KRUEGER T, DEWALD A. Cujo: efficient detection and prevention of drive-by-download attacks[C]//Proceedings of Proceedings of the 26th Annual Computer Security Applications Conference. New York: ACM Press, 2010: 31-39.

[28] FASS A, KRAWCZYK R P, BACKES M, et al. JaSt: fully syntactic detection of malicious (obfuscated) JavaScript[C]//Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2018: 303-325.

[29] GENCER K, BAŞÇIFTÇI F. Time series forecast modeling of vulnerabilities in the android operating system using ARIMA and deep learning methods[J]. Sustainable Computing: Informatics and Systems, 2021, 30: 100515.

[30] GRUSKA N, WASYLKOWSKI A, ZELLER A. Learning from 6, 000 projects: lightweight cross-project anomaly detection[C]//Proceedings of the 19th International Symposium on Software Testing and Analysis. New York: ACM Press, 2010: 119-130.

[作者简介]



秦佳伟（1993− ），男，满族，辽宁本溪人，北京邮电大学博士生，国家计算机网络应急技术处理协调中心工程师，主要研究方向为移动端安全分析、物联网安全分析等。

张华（1978− ），女，吉林四平人，博士，北京邮电大学副教授，主要研究方向为网络安全、隐私保护等。

严寒冰（1975− ），男，江西进贤人，博士，国家计算机网络应急技术处理协调中心教授级工程师，主要研究方向为网络安全、计算机图形学等。

何能强（1985− ），男，浙江义乌人，博士，国家计算机网络应急技术处理协调中心高级工程师，主要研究方向为移动恶意程序分析、应用程序安全检测等。

涂腾飞（1990− ），男，山东临沂人，博士，北京邮电大学在站博士后，主要研究方向为网络安全、移动安全等。