基于双向 LSTM 的 Java 开源软件漏洞检测

刘嘉华[1] 万 明[2*] 周晨曦[2] 张 攀[3]

[1](南瑞集团有限公司(国网电力研究院有限公司) 江苏 南京 211106)

[2](南京南瑞信息通信科技有限公司 江苏 南京 210009)

[3](国家电网有限公司信息通信分公司 北京 100761)

摘 要 开源(Open Source)软件的使用、修改和分发不受许可证的限制,但研究表明其往往存在诸多安全漏洞。开发人员对开源软件的使用会直接影响到自身软件的安全和质量。提出了基于双向 LSTM 的 Java 开源软件漏洞检测方法:将函数抽象为包含数据依赖关系和控制依赖关系的中间表示;将中间表示映射为向量并贴上标签;运用双向 LSTM 训练出漏洞检测模型。实验结果表明,漏洞检测结果的准确率和召回率分别达到 93.8% 和 90.1%,能够较为准确地检测到 Java 开源软件中的安全漏洞。

关键词 开源软件 漏洞检测 中间表示 双向 LSTM

中图分类号 TP305 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2020.12.051

VULNERABILITY DETECTION IN JAVA OPEN SOURCE SOFTWARE BASED ON BIDIRECTION LSTM

Liu Jiahua[1] Wan Ming[2*] Zhou Chenxi[2] Zhang Pan[3]

[1](*Nari Group Corporation/State Grid Electric Power Research Institute, Nanjing 211106, Jiangsu, China*)

[2](*Nanjing NR Information and Communication Technology Co., Ltd., Nanjing 210009, Jiangsu, China*)

[3](*State Grid Information and Telecommunication Co., Ltd., Beijing 100761, China*)

Abstract The use, modification, and distribution of open source software are also not subject to license restrictions. However, research shows that open source software often has many security vulnerabilities. For developers, the application of open source software directly affects the security and quality of their software. This paper proposes a Java open source software vulnerability detection method based on bidirectional LSTM. It abstracted the function into an intermediate representation containing data dependencies and control dependencies, then mapped the intermediate representations to vectors and labeled them, and finally used bidirectional LSTM to train a vulnerability detection model. The experiment shows that the accuracy and recall rates of the vulnerability detection results can reach 93.8% and 90.1% respectively, which demonstrates the proposed approach can detect the security vulnerabilities in Java open source software accurately.

Keywords Open source software Vulnerability detection Intermediate representation Bidirectional LSTM

0 引 言

随着计算机技术应用的不断深化,软件的需求和规模不断增加,给开发人员带来了新的挑战。有效地使用开源软件可以提高软件开发效率,降低软件开发成本。为了更快速高效地完成开发,开源软件得到了推崇。数据显示,2018 年各大平台的工具包的下载量均有了不同程度的上涨,仅 npm 在 2018 年全年的下载次数就达到了令人难以置信的 3 170 亿次。

开源软件的广泛使用在给开发人员提供极大便利的同时,也带来了很多问题。(1) 开源软件漏洞与日

收稿日期:2019-07-02。国家电网公司总部科技项目(SGFJXY00YJJS1800074)。**刘嘉华**,高工,主研领域:电力信息化技术。**万明**,高工。**周晨曦**,工程师。**张攀**,工程师。

俱增。2018 年,npm 的漏洞数量增长了 47%,RHEL、Debian 和 Ubuntu 增加了 4 倍多。两年内开源软件的漏洞数量增长了 88%。(2) 开源软件漏洞得不到重视和修复。37% 的开源开发者在持续集成(Continuous Integration,CI)期间没有实施任何类型的安全测试。54% 的开发者没有对 Docker 镜像进行任何安全测试。从漏洞添加至开源软件包到修复漏洞的时间中位数超过 2 年。(3) 开源软件的使用引入了更多的漏洞。78% 的漏洞来源于对开源软件直接或间接的引用[1]。因此,针对开源软件的漏洞检测有着极其重要的意义。

漏洞检测技术主要分为静态分析[2-5]和动态分析[6-8]两类。随着人工智能的兴起,研究人员开始尝试将传统技术与机器学习结合起来。现有的基于机器学习的方法多用于软件缺陷预测,但与软件缺陷相比,项目内含有的漏洞数量更少,因此检测难度更大。Shin 等[9]利用人工神经网络识别二进制代码中的函数,他们将二进制代码根据字节码进行划分,对采集到的 2 200 个二进制代码进行训练和实验,结果表明采用递归神经网络能够更高效、更准确地识别出二进制代码中的函数。HSOMiner 是一种基于机器学习的程序分析技术[10],能够快速高效地发现移动应用中隐藏的敏感行为。Chowdhury 等[11]从代码复杂度、耦合度和内聚度等角度对程序模块进行度量。

本文以静态分析为基础,融合抽象语法树、数据流、控制流分析的结构化信息形成中间表示,并运用较前沿的双向 LSTM 神经网络学习其隐含的漏洞模式以检测漏洞。双向 LSTM 是传统 LSTM 的扩展,可以提高序列分类问题的模型性能。在输入序列的所有时间步长可用的问题中,双向 LSTM 在输入序列上训练两个而不是一个 LSTM。输入序列中的第一个是原有的,第二个是输入序列的反转副本。这可以为网络提供额外的上下文,从而更快、更充分地学习问题。对比实验结果证明了该方法的有效性和可行性。

1　方案设计

1.1　实例分析

设计漏洞检测方案之前,首先需要对漏洞做一个简单的分析。以 CWE(Common Weakness Enumeration)[12]列表中较为常见的数组索引的不正确验证(Improper Validation of Array Index)漏洞为例,漏洞的产生需要满足两个必要条件:不安全的数据和缺少自定义的数据验证,二者缺一不可。

图 1 所示的一段代码在运行时显然会发生数组索引溢出。但是只要任意破坏其中某一条件,能够确保数据安全(见图 2)或者拥有自定义的数据验证(见图 3),则该漏洞不复存在。

```
int data;
data = (new SecureRandom( )).nextInt( );
int array[ ] = { 0, 1, 2, 3, 4 };
IO.writeLine(array[data]);
```

图 1　数组索引溢出

```
int data;
data = 2;
int array[ ] = { 0, 1, 2, 3, 4 };
IO.writeLine(array[data]);
```

图 2　数据安全

```
int data;
data = (new SecureRandom( )).nextInt( );
int array[ ] = { 0, 1, 2, 3, 4 };
if (data >= 0 && data < array.length)
    IO.writeLine(array[data]);
else
    IO.writeLine("error")
```

图 3　自定义的数据验证

经过上述分析,要想确保漏洞检测的准确率,提取的源代码特征中需要同时包含方法的数据和控制依赖关系。因此,本文在使用静态分析提取源代码特征的过程中引入了数据流和控制流分析。

1.2　框架设计

本文提出了一种 Java 开源软件漏洞检测方法,其框架设计如图 4 所示。

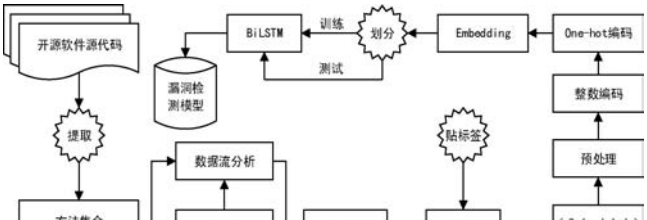

图 4　框架设计

该方法主要分为两个部分:特征提取和模型训练。特征提取部分先为方法生成抽象语法树(Abstract Syntax Tree,AST),依据节点类型提取关键变量,该类变量或接收外部输入,有可能是污染源,或位于外部输入的数据流传播路径上,有可能被污染。在此基础上,运用数据流分析变量与语句、方法间的数据依赖关系,并将数据类型替换为完整类型,如将 String s 替换为 java.lang.String s。然后,运用控制流分析获取方法内各语句间的控制(条件判断、循环等)依赖关系,结合数据流分析的结果生成控制流图(Control Flow Graph,CFG),

保存方法的控制流信息。最后,遍历生成的控制流图,生成包含数据和控制依赖关系的中间表示。

模型训练部分先为中间表示贴上“0”(没有安全漏洞)或“1”(含有安全漏洞)标签,生成 < Code, Label > 对的集合。接着,对 Code 字段做分词处理,将生成的中间表示转换为 token 序列,并为每个 token 从 1 开始分配一个连续的唯一的整数索引。然后,将原序列中的每个 token 替换为其对应的整数编码。再将每个整数按照词表的大小展开,索引位置 1,其余位填 0,形成 One-hot 编码。考虑到 One-hot 矩阵过于稀疏,在搭建神经网络的过程增设 Embedding 层以压缩向量,为矩阵降维。最后,将数据集划分为训练集和测试集,运用双向 LSTM 训练出漏洞检测模型。

2 特征提取

2.1 抽象语法树

抽象语法树是源代码的一种树状的表现形式,树上的每个节点都表示源代码中的一种结构,如图 5 所示。遍历抽象语法树,根据节点类型可以非常容易地提取出源代码中的关键变量,同时对用户自定义的变量名做统一替换。例如,同样是 bool 类型的变量,程序员 A 命名为 isTrue,程序员 B 命名为 isRight,程序员 C 命名为 exist 等,在遍历抽象语法树的过程中将统一被替换为 b,同一方法中的不同 bool 类型变量用 b1、b2 区分。本实验中,抽象语法树的实现基于开源工具 ANTLR4[13]。

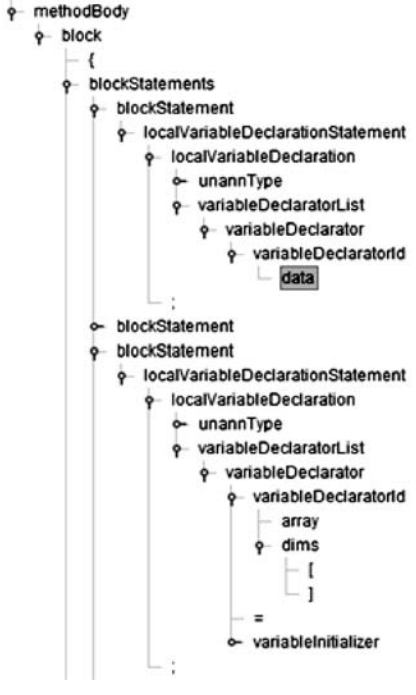

图 5 抽象语法树

2.2 控制流图

控制流图是一个方法或程序的抽象表现,代表了一个程序执行过程中会遍历到的所有路径。它用图的形式表示一个过程内所有基本块执行的可能流向。

图 6 为图 3 中的代码示例生成的控制流图,该图由 Soot[14]开源工具生成。抽象语法树分析为方法选取关键节点,确定数据流分析的需求对象。数据流分析在方法间追踪该对象的数据流向,并为其添加数据依赖关系,如将 SecureRandom 扩充为 java. security. SecureRandom。为了更好地追踪数据的依赖关系,数据流分析过程中会产生一些以“ $ ”开头的用于过渡的临时变量。控制流分析以这些数据作为节点,将数据流分析所得的数据依赖关系保存在节点内,分析节点在方法内部的控制依赖关系,并用有向边表示出来,继而生成了控制流图。

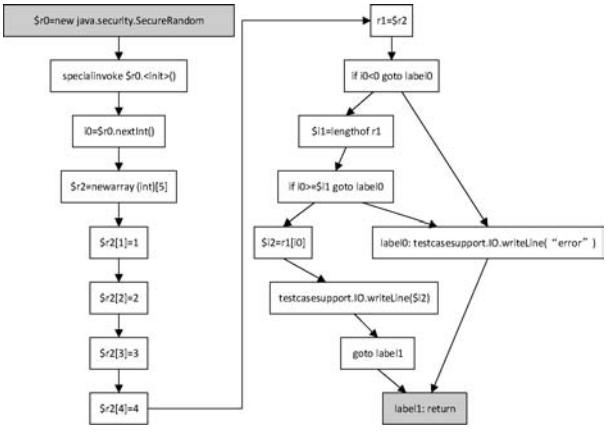

图 6 控制流图

2.3 中间表示

遍历上述控制流图可以生成如图 7 所示的基于 Jimple 的代码的中间表示。

```
1  java.security.SecureRandom $r0;
2  int i0, $i1, $i2;
3  int[ ] r1, $r2;
4  $r0 = new java.security.SecureRandom;
5  specialinvoke $r0.<java.security.SecureRandom:void <init>( )>( );
6  i0 = virtualinvoke $r0.<java.security.SecureRandom:int nextInt( )>( );
7  $r2 = newarray(int)[5];
8  $r2[1] = 1;
9  $r2[2] = 2;
10 $r2[3] = 3;
11 $r2[4] = 4;
12 r1 = $r2;
13 if i0 < 0
14    goto label0;
15 $i1 = lengthof r1;
16 if i0 >= $i1
17    goto label0;
18 $i2 = r1[i0];
19 staticinvoke<testcasesupport.IO:void writeLine(int)>($i2);
20 goto label1;
21 label0:
22    staticinvoke<testcasesupport.IO:void writeLine(java.lang.String)>("error");
23 label1:
24    return;
```

图 7 中间表示

将代码处理成如图 7 所示的中间表示主要有以下几点好处:(1) 简化了 Java 语句,将原来复杂的语句简化为 15 种基本语句,抽象程度较高;(2) 保留了变量的类型;(3) 对变量名做了统一替换,压缩了词表的大小。

3　模型训练

3.1　预处理

预处理的工作主要分为以下几个部分:(1) 根据空格对生成的中间表示做分词处理,将每条方法转化成 token 序列。(2) 为了将向量转换为固定的维度以供训练,需要筛除长度超过 500 的 token 序列,并将长度不足 500 的序列以某一特殊填充符填充至 500(在样本长度的分布统计中,绝大多数样本的 token 序列长度均在 500 以内,超过 500 的样本所占比重不到 5%。预处理需要将所有的 token 序列填充至相同长度,如果迁就少数过长样本,将导致序列过长且填充大量无意义填充符,因此筛除此部分样本,该操作不会影响实验结果的准确率和召回率)。(3) 建立词表,将所有 token 不重复地加入到词表中,从 1 开始为词表中的每个 token 分配一个连续的唯一的整数索引。(4) 将所有 token 替换为它对应的索引值。

3.2　One-hot 编码

One-hot 编码,又称为独热编码、一位有效编码,其方法是采用 N 位状态寄存器来对 N 个状态进行编码,每个状态都有它独立的寄存器位,并且在任意时候只有一位有效。预处理部分将 token 序列转化为整数序列其实就是在为转化为 One-hot 编码做准备。以 token 序列表示的文本虽然可读性比较好,却无法用深度学习,因此我们需要对其进行特征数字化,拟采用的方法就是转换为 One-hot 编码。这样做的好处一是解决了分类器不好处理离散数据的问题,二是在一定程度上起到了扩充特征的作用,三是解决了 token 长度不一致的问题。

当然该编码的缺点也很明显:(1) 这是一个词袋模型,不考虑词与词之间的顺序信息,该缺点可以通过选取双向 LSTM 学习顺序信息来有效解决。(2) 它假设词与词之间相互独立,虽然在大多数情况下,词与词之间是相互影响的,但是由于该实验的研究对象是源代码,恰好符合这一情境,进而转变为一种优势。(3) 它得到的特征是离散稀疏的,因此需要在模型中加入 Embedding 层对向量进行压缩降维。

One-hot 编码的实现是基于整数编码的结果,将每个整数按照词表大小的长度展开,索引位置 1,其余位置 0。

3.3　Embedding

虽然变量名的替换在一定程度上对词表进行了压缩,但是随着项目数量的不断增多,词表的大小还是很容易就能达到万的级别,这意味着一个 token 序列展开后的长度是百万级的。因此,需要在模型中增加 Embedding 层对 One-hot 编码进行压缩。其原理如图 8 所示。

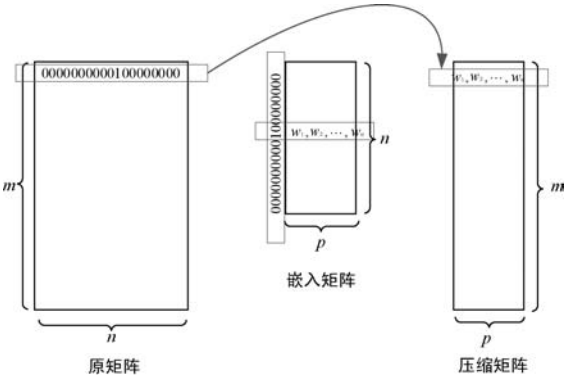

图 8　Embedding 原理

原矩阵是一个 $m \times n$ 的矩阵,m 是样本个数,n 是压缩前向量的维度,长度等于词表的大小。引入一个 $n \times p$ 的嵌入矩阵,p 是压缩后向量的维度。由于 One-hot 向量的特殊性,将原矩阵中的某一条向量与嵌入矩阵做乘法就相当于从嵌入矩阵中取出了某一行,而该行即为原向量的压缩向量。经过 Embedding 之后,原来大小为 $m \times n$ 的矩阵被压缩为大小为 $m \times p$ 的压缩矩阵。

3.4　神经网络

为了更好地学习代码的序列信息,实验选取了双向 LSTM 神经网络用于训练。双向 LSTM 组合了前向 LSTM 与后向 LSTM,所以可以包含更多的信息。该神经网络的架构如图 9 所示。

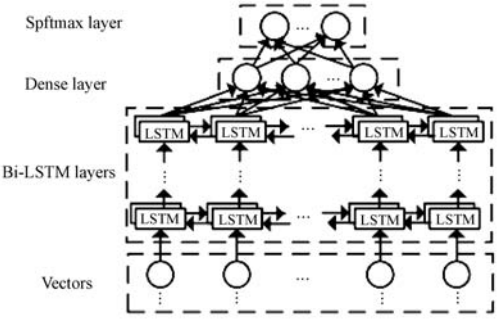

图 9　双向 LSTM

4　实验评估

为了对本文提出的方法做进一步的分析,该部分从最新的基于机器学习做漏洞检测的成果中选取了两个较为相似的工作 VulDeePecker[16] 与 AE-KNN[17] 进行了对比实验。实验的计算机环境为:处理器为 Intel Core i7-4790 3.60 GHz,内存 32 GB,硬盘 1 TB,GPU Geforece GTX 1070 Ti。

4.1 数据集

该实验采用的数据集来源于美国国家标准与技术研究院(NIST)的软件保障参考数据集(SARD)[15]。该数据集包含了对各种漏洞的测试用例,对应于 CWE 列表中 112 种安全漏洞类型。以 CWE129(数组索引的不正确验证)为例,其统计信息如表 1 所示。数据集按照 8∶2划分训练集和测试集。

表 1 CWE129 数据集统计

安全方法	不安全方法	无关方法	总数
13 916	7 285	4 402	25 603

4.2 评估指标

本实验以准确率和召回率来衡量方法的检测结果。相关计算公式如下:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

式中:*TP* 表示模型判断方法存在安全漏洞且漏洞真实存在;*FP* 表示模型判断方法存在安全漏洞而漏洞真实不存在;*TN* 表示模型判断方法不存在安全漏洞且漏洞真实不存在;*FN* 表示模型判断方法不存在安全漏洞而漏洞真实存在。FP 和 FN 均属于误判。Precision 为准确率,表示模型判断出真实存在安全漏洞的方法占全部存在安全漏洞方法的比例。Recall 为召回率,表示模型判断出的含有漏洞的方法数占全部不安全方法数的比例。F-measure 是 Precision 和 Recall 的调和平均数,是一种对准确率和召回率进行综合考量的指标。

4.3 参数设置

实验在 GPU 上训练模型,使用随机梯度下降(Stochastic Gradient Descent,SGD)来训练和更新参数。批处理大小设置为 128,LSTM 隐藏状态和词嵌入的维度设置为 200。参数梯度的上限设置为 5,为避免过拟合现象的发生,设置 dropout 为 0.3。具体参数配置如表 2 所示。

表 2 实验参数设置

参数	值	参数	值
学习速率	0.001	嵌入大小	200
最大训练样本	20 万	隐含层节点数	128
批处理大小	128	神经网络层数	2
词汇表大小	14 860	最大梯度	5

4.4 实验结果

本文首先人为地观察模型的检测结果,以做初步评估。由于在源代码的中间表示中引入了数据和控制依赖,因此即使是对于两个差异性极小的代码段,模型依然能够准确地做出区分。如图 10 所示的两段源代码尝试从用户指定的值构建列表,同时引入了自定义的数据验证以检查传入的 untrustedListSize 参数,函数名、参数甚至结构完全相同。但是,在上段代码中,如果提供了 0 值,代码将构建一个大小为 0 的数组,然后尝试在第一个位置存储一个新的 Widget,从而引发数组越界。本实验中,模型能够有效地对该类情况做出分类,这是许多传统的漏洞检测方法所不具备的。

```
1 private void buildList(int untrustedListSize){
2   if(0 > untrustedListSize){
3     die("Unsafe value supplied for list size.");
4   }
5   Widget[] list = new Widget[untrustedListSize];
6   list[0] = new Widget();
7 }
```

```
1 private void buildList(int untrustedListSize){
2   if(0 >= untrustedListSize){
3     die("Unsafe value supplied for list size.");
4   }
5   Widget[] list = new Widget[untrustedListSize];
6   list[0] = new Widget();
7 }
```

图 10 检测示例

为了进一步验证该方法的有效性,本文基于 SARD 数据集与现有的两个工作 VulDeePecker 和 AE-KNN 做了对比实验。实验结果如表 3 所示。

表 3 不同方法实验结果对比 %

模型	Precision	Recall	F-measure
VulDeePecker	91.7	82.0	86.6
AE-KNN	93.5	87.8	90.6
本文方法	93.8	90.1	91.9

VulDeePecker 首先提取源代码中的 API 和库方法调用,运用代码切片将源代码分割成一个个 code gadgets。然后用统一的标识符替换掉用户自定义的变量名和函数名,并为每个 code gadgets 贴上标签。最后选取双向 LSTM 在数据集上训练出相应模型。但是该方法忽略了部分控制流信息,因此在准确率和召回率上稍显逊色。

AE-KNN 首先应用图形数据库形成源代码的代码属性图,然后依据某种规则遍历代码属性图形成 API 序列并量化为特征向量,最后对特征向量进行聚类,提取每一类中异常值排序高的样本函数匹配漏洞库,得到代码中的漏洞。

实验结果表明,本文方法在准确率和召回率上均优于对比方法。

5 结　语

为检测开源软件中存在的大量漏洞,本文提出了一种基于双向 LSTM 的 Java 漏洞检测方法。首先运用静态分析提取源代码语义特征并生成中间表示,然后在将生成的中间表示映射为向量的同时为其贴上“是否安全”的标签,最后运用神经网络在数据集上训练生成漏洞检测模型。静态分析主要包括抽象语法树、数据流和控制流分析,目的是提取方法完整的数据和控制依赖,以提高漏洞检测的准确性。神经网络选择双向 LSTM,目的是可以更好地学习源代码的序列信息。

实验结果表明,模型在测试集上取得了 93.8% 的准确率和 90.1% 的召回率,均优于现有的基于机器学习的漏洞检测方法,验证了本文方法的优越性。后续工作主要分为两个方面,一是进一步提升模型的精确度,减少人工参与,实现开源软件漏洞的自动化检测;二是尝试将本文方法扩展到除 Java 外的其他语言,验证其适用性。

参 考 文 献

[1] Snyk. The state of open source security report[R/OL]. [2019-07-02]. https://bit.ly/SoOSS2019.

[2] Shankar U, Talwar K, Foster J S, et al. Detecting format string vulnerabilities with type qualifiers[C]//Proceedings of the 10th conference on USENIX Security Symposium. ACM, 2001, 10: 16.

[3] Yamaguchi F, Golde N, Arp D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//2014 IEEE Symposium on Security and Privacy. IEEE, 2014: 590 - 604.

[4] Qian C X, Luo X P, Le Y, et al. VulHunter: Toward discovering vulnerabilities in android applications[J]. IEEE Micro, 2015, 35(1): 44 - 53.

[5] Eschweiler S, Yakdan K, Gerhards-Padilla E. DiscovRE: Efficient cross-architecture identification of bugs in binary code[C]//The Network and Distributed System Security Symposium(NDSS), 2016.

[6] Li Y K, Chen B H, Chandramohan M, et al. Steelix: Program-state based binary fuzzing[C]//2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017: 627 - 637.

[7] Wang J J, Chen B H, Wei L, et al. Skyfire: Data-driven seed generation for fuzzing[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 579 - 594.

[8] Blanda A. Fuzzing Android: a recipe for uncovering vulnerabilities inside system components in Android[C]//Black Hat Europe, 2015.

[9] Shin Y, Williams L. Can traditional fault prediction models be used for vulnerability prediction? [J]. Empirical Software Engineering, 2013, 18(1): 25 - 59.

[10] Pan X R, Wang X Q, Duan Y, et al. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps[C]//Network and Distributed System Security Symposium, 2017.

[11] 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.

[12] The MITRE Corporation. Common weakness enumeration? [DB/OL]. [2019-07-02]. https://cwe.mitre.org.

[13] Parr T, Harwell S, Vergnaud E, et al. ANTLR4[CP/OL]. [2019-07-02]. https://github.com/antlr/antlr4.

[14] Arzt S, olhotak, mbenz89, et al. Soot[CP/OL]. [2019-07-02]. https://github.com/Sable/soot.

[15] National Institute of Standards and Technology. NIST software assurance reference dataset[DS/OL]. [2019-07-02]. https://samate.nist.gov/SARD.

[16] Li Z, Zou D Q, Xu S H, et al. VulDeePecker: A deep learning-based system for vulnerability detection[C]//The Network and Distributed System Security Symposium, 2018.

[17] 李元诚,黄戎,来风刚,等. 基于深度聚类的开源软件漏洞检测方法[J]. 计算机应用研究, 2020, 37(4): 1107 - 1110, 1114.

(上接第 308 页)

[29] 李晓峰,赵海,王家亮,等. 基于增加一个随机数的 ElGamal 数字签名算法的改进[J]. 东北大学学报(自然科学版), 2010, 31(8): 1102 - 1104, 1112.

[30] 喻秋叶. 生日悖论在密码学中的应用[D]. 武汉:华中师范大学, 2013.

[31] 张先红. 数字签名原理及技术[M]. 北京:机械工业出版社, 2004: 95 - 95.

[32] 韩益亮,杨晓元,户军茹,等. 改进的 ECDSA 签名算法[C]//第二十届全国数据库学术会议论文集(研究报告篇), 2003.

(上接第 315 页)

[18] Lin Q, Yang L, Guo Y. Proactive batch authentication: Fishing counterfeit RFID tags in muddy waters[J]. IEEE Internet of Things Journal, 2019, 6(1): 568 - 579.

[19] Rashid N, Choudhury S, Salomaa K. Localized algorithms for redundant readers elimination in RFID networks[J]. International Journal of Parallel, Emergent and Distributed Systems, 2019, 34(3): 260 - 271.

[20] Corchia L, Monti G, Tarricone L. A frequency signature RFID chipless tag for wearable applications[J]. Sensors, 2019, 19(3): 494.