计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(4):9−17 [doi: 10.15888/j.cnki.csa.006848]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于深度学习的模糊测试种子生成技术①

李张谭[1,2], 程　亮[2], 张　阳[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院软件研究所 可信计算与信息保障实验室, 北京 100190)

摘　要: 模糊测试被广泛应用于各种软件和系统的漏洞挖掘中. 而模糊测试的效果与其采用的变异策略以及初始种子文件的代码覆盖率有直接的关系. 本文提出了一种基于深度学习的种子文件生成方法, 分析并学习初始种子文件和其在目标程序中的执行路径之间的关系, 最终输出可能覆盖新执行路径的种子文件, 从而提高初始种子文件集合的代码覆盖率. 我们以 PDF 阅读器作为目标程序进行了实验, 实验结果表明该方法所生成的种子文件保证了良好的通过率, 而且明显提高了代码覆盖率. 同时实验证明该方法在针对多种 PDF 阅读器进行模糊测试时都获得了更高的代码覆盖率.

关键词: 模糊测试; 深度学习; 文本生成; 代码覆盖; seq2seq 模型

引用格式: 李张谭,程亮,张阳.基于深度学习的模糊测试种子生成技术.计算机系统应用,2019,28(4):9–17. http://www.c-s-a.org.cn/1003-3254/6848.html

Seed Generation for Fuzzing Based on Deep Learning

LI Zhang-Tan[1,2], CHENG Liang[2], ZHANG Yang[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Fuzzing is widely used for different kinds of software and systems to detect the vulnerabilities. The effectiveness and efficiency of fuzzing is related to the mutation strategy of the seed files and the code coverage of the seed files for the target program. This study proposes a new method based on deep learning for seed generation. The proposed method analyses and learns the correlation between the seed files and their paths in the target program. Finally, the proposed method generates seed files that more likely explore uncovered paths, thus increases the code coverage of the initial seed files for the target program. Aiming at the PDF reader, we carry out the experiment. The results demonstrates that the seed files generated by proposed method have a good passing rate of the PDF reader, in the meantime, significantly improve the code coverage. The experiment also indicates the applicability of proposed method: the seed files which are generated for specific target program (PDF reader) can also obtain higher code coverage when fuzzing some other kinds of PDF readers.

Key words: fuzzing; deep learning; text generation; code coverage; seq2seq model

1　引言

模糊测试作为目前漏洞挖掘中使用最广泛、最有效的方法, 其本质是通过一定的策略, 构造出非预期的输入, 然后监控目标程序或者软件的异常结果 (比如说

① 基金项目: 国家自然科学基金 (61471344, 61772506); 国家重点研发计划 (2017YFB0802902)
Foundation item: National Natural Science Foundation of China (61471344, 61772506); National Key Research and Development Program of China (2017YFB0802902)
收稿时间: 2018-10-10; 修改时间: 2018-10-30; 采用时间: 2018-11-14; csa 在线出版时间: 2019-03-28

崩溃) 来发现程序或者软件的漏洞. 由于模糊测试可以把手工测试转为高度的自动化测试, 该技术被安全人员广为使用, 国外一些 IT 巨头比如微软、谷歌也把模糊测试技术作为软件质量保证的核心技术之一, 成立了相应的小组, 进行相关的研发工作. 模糊测试已经被广泛应用于网络协议、操作系统内核、浏览器、图像处理等各种软件的漏洞检测中. 作为最有名的模糊测试工具之一, AFL(American Fuzzy Lop) 从 2013 年推出之后, 发现了 143 个软件中的超过 400 个漏洞[1].

模糊测试的效率很大程度上取决于生成的非预期输入是否能够有效地提高其在目标程序中的代码覆盖率. 而目前的模糊测试工具一般通过对预先准备好的“种子”进行变异, 来生成新的输入. 由于种子文件的复杂性, 为了生成有效的恶意输入, 可能需要几百万次的变异. 可以说模糊测试的过程就相当于是去搜索一个高质量的变异集合, 以得到更高的代码覆盖率和更多的崩溃. 一般从对目标程序的认知角度可以把模糊测试分成 3 大类: 黑盒、白盒以及灰盒测试. 这三种不同的测试方法也对应着不同的变异策略. 黑盒测试不知道目标程序的内部结构, 所以把目标程序当成黑盒来进行测试, 可能用到一些简单的随机测试的方法. 白盒测试指的是在完全了解目标程序的条件下, 利用程序分析的方法进行变异, 针对性地提高代码覆盖率或者覆盖特定的程序模块. 灰盒测试一般使用一些手段 (比如插桩) 来获取一些模糊测试信息, 比如基本块的转移等, 来有效地指导变异策略. 所以为了提高新的输入在目标程序上的代码覆盖率, 基本上有两种途径: 一种是改进变异策略, 另外一种是提高“种子”的质量 (即提高种子本身在目标程序中的代码覆盖率).

通过改进变异策略来提高种子输入的代码覆盖率对于拥有复杂格式的文件而言比较困难[2], 通过一般的变异策略生成的种子文件可能都无法通过文件解析器的语法检查. 所以针对复杂格式文件进行模糊测试, 提高种子输入本身的质量可能是更好的选择. 有很多相关的工作[2,3]使用语法概率模型、机器学习相关的模型来学习种子文件的语法结构. 但是, 这些工作都没能有效地利用种子输入的初始路径.

针对现有工作较少考虑利用种子输入的初始路径来提高种子输入的代码覆盖率的现状, 本文提出了一种新的框架, 通过机器学习来挖掘初始种子和其对应执行路径的关系, 然后利用这种关系去指导生成新的种子进而触发更多的新路径. 我们以比较复杂的 PDF 文件结构为研究对象, 并且在我们的框架里引入了两个模型.

第一个模型基于马尔可夫链模型. 该模型通过学习 PDF 文件的执行路径生成概率模型, 进而生成可能的新的执行路径. 新生成的路径用来给 PDF 的生成做指导.

第二个模型是 sequence-to-sequence 网络 (RNN 的一种变形). 该网络通过训练 PDF 对应的执行路径和 PDF 文本语料的关系, 以第一个模型所生成的新路径作为输入, 输出新的 PDF 文件.

我们在 CAJViewer(中国知网官方提供的中文期刊 PDF 阅读器) 上做了一系列的实验. 需要说明的是, 我们选择 CAJViewer 作为我们主要的研究对象是因为, 相对于其他一些主流的 PDF 阅读器 (比如 Adobe、Foxit) 我们可以更加方便地使用工具获取到 PDF 文件的执行路径. 实验表明, 我们的方法相比较目前其他模糊测试方法明显地提高了原始种子输入的代码覆盖率, 同时我们的方法生成的新 PDF 种子在保证了一定的通过率的情况下, 还生成了很多导致程序崩溃的种子输入.

本文的主要贡献如下:

(1) 我们提出了一个基于深度学习的种子生成框架, 利用神经网络学习 PDF 文件的语法结构以及种子的执行路径和种子文本的关系, 最终输出可能覆盖新执行路径的种子文件. 实验表明针对不同的 PDF 阅读器, 我们的方法提高了 0.35%–2.55% 的代码覆盖率.

(2) 我们提出了使用马尔科夫链模型来生成新的执行路径指导种子文件的生成. 实验表明该模型可以生成质量较高的执行路径序列.

文章的后续结构如下, 第 2 节将具体介绍相关研究工作与背景, 第 3 节将详细介绍我们的框架细节, 第 4 节会具体阐述我们用到的两个模型, 第 5 节是具体的实验评估, 最后第 6 节和第 7 节是讨论和总结.

2 相关研究与背景

2.1 相关工作

在已有的相关工作中, 针对提高变异策略的相关研究可以被分为: 基于程序和基于模糊测试本身. 基于程序的方法有利用符号执行[4–7]或者其他的程序分析技术[8–11]来挖掘输入的种子与目标程序中相关代码的关系, 然后利用得到的关系去生成相应的种子以探索相关的代码. 另一方面, 有一些基于模糊测试本身的方

法[12−15], 这些工作从已执行的模糊测试中学习了一些经验, 并利用这些经验来进一步提高种子的变异机制.

但是仅仅通过改变变异策略来提高覆盖率有一定的局限性, 最明显的缺点是效率低下, 无法明显提高新生成的输入的代码覆盖率[2]. 其中有一部分原因是因为有些种子文件的语法比较复杂, 不太容易设计出对应的变异策略来生成高质量的输入, 一般的变异策略所生成的输入文件可能都无法通过软件初步的语法或者语义检查.

因此, 很多方法就以提高种子的质量作为切入点. 文献[2]利用语法树来构建一个语法概率模型进而学习种子文件的语法结构, 来提高新生成的种子的质量. 他们以 XSLT 和 XML 解析器作为目标程序, 把生成的新种子文件放入 AFL 里做模糊测试, 提高了目标程序 20% 的代码覆盖率. 另外有利用机器学习相关的模型算法来学习种子文件的内部语法结构[3], 他们以更加复杂的文件结构 PDF 作为研究对象, 通过循环神经网络 RNN(Recurrent Neural Network) 从庞大的语料库中学习种子文件的语法和语义, 最后生成相对质量较高的种子文件. 文献[16]利用 RNN 网络对 AFL 进行改进, 并对 PDF 以及 PNG 等多种格式的文件进行了测试, 实验表明他们的方法对于简单的文件格式的效果要更好一些. 但是, 上述这些方法都没有考虑到种子输入和这些种子对应的执行路径之间的关系. 所以他们可能经常生成一些包含了相同执行路径的种子. 在文献[2]中只有少于三分之一的新种子被 AFL 认为是有用的.

2.2 背景介绍

我们以 PDF 文件作为研究对象是因为 PDF 相对于其他一些文件格式, 比如图片、脚本语言等, PDF 的语法更加复杂. 所以对于 PDF 文件的模糊测试, 保证生成的种子文件的通过率和多样性变得更加困难.

因为 PDF 文件格式比较复杂, 所以在这里先简单地介绍一下 PDF 这种文件格式. 目前 PDF 的官方文档对 PDF 文件的定义是由以下四部分组成, 具体如图 1.

```
2 0 obj
 <</Type /Pages
   /Kids [3 0 R]
   /Count 1
 >>
endobj
```

(a) 文件体

```
xref
0 4
00000000000 65535 f
00000000010 00000 n
00000000069 00000 n
00000000141 00000 n
```

(b) 交叉引用表

```
trailer
<< /Root 1 0 R
/Size 4
>>
startxref
249
%%EOF
```

(c) 文件尾

图 1 PDF 文件的主要组成部分

(1) 文件头: 定义了 PDF 的版本, 一般而言是一行字符串, 比如“%PDF-1.7”.

(2) 文件体: 包含了一系列的 obj 对象, 表示 PDF 的文本内容. 如图 1(a) 所示, 一个 obj 对象由两部分组成: 1) 第一部分是 obj 对象的标识符, 包括表示开始的“2 0 obj”和表示结束的“endobj”. 其中第一个数字称为对象号, 用来唯一标识一个对象, 比如 2 表示第二个 obj 对象. 第二个数字是生成号, 用来表明 obj 对象在被创建后的第几次修改. “0”表示创建后没有被修改过. 2) 第二部分是 obj 对象的内容. 一个 obj 包含了 8 种类型, 包括布尔型、数值型、字符型、数组型、字典型、名称、数据流型、和 NULL. 其中布尔型、数值型、字符型、数组型、字典型和编程语言中对应的类型一样. 数组型和字典型分别需要“[]”和“<<>>”括起来. 名称是在“/”后面标识的字符串, 用来标识对应的关键字. 数据流一般是一大段数据字符, 用关键字“stream”和“endstream”前后标识. 图 1(a) 中的“/Type/Pages”说明该 obj 表示的是一页 page; “/Count 1”表示该页有一个子 obj 对象; “/Kids [3 0 R]”表示这个子 obj 的对象号是 3, 它的生成号是 0.

(3) 交叉引用表: 用来快速访问 PDF 中的 obj 对象. 交叉引用表以 xref 开头, 以图 1(b) 为例, 第一行“0 4”说明了下面各行所描述的对象号是从 0 开始, 并且有 4 个对象. 一般每个 PDF 文件都是以“0000000000 65535 f”这一行开始交叉引用表的, 说明对象 0 的起始地址为 00000000000, 生成号为 65535, 65535 也是最大生成号, 不可以再进行更改, 而最后的 f 表明该对象为 free(这个对象可以看作是文件头); “0000000010 00000 n”表示对象 1, 00000000010 是其偏移地址, 00000 为生成号, 表明该对象从未被修改过, n 表示该对象在使用, 区别于自由对象, 可以更改; 后面几行也是类似的.

(4) 文件尾: 包含了一些 PDF 的其他信息, 比如说交叉引用表的位置等. 以图 1(c) 为例, trailer 表示文件尾的开始; “/Root 1 0 R”说明根对象的对象号为 1; “/Size 4”说明该 PDF 文件的对象数目. “startxref 249”表示交叉引用表的偏移地址, 进而可以找到 PDF 文档中所有对象的相对地址, 从而访问对象.

3 框架概述

我们的系统框架如图 2 所示. 其中主要的模块有路径记录器、路径生成器、PDF 解析器、PDF 生成

器. 该框架利用机器学习的方法来提高生成的种子的质量 (代码覆盖率), 整个框架主要分为 3 个部分.

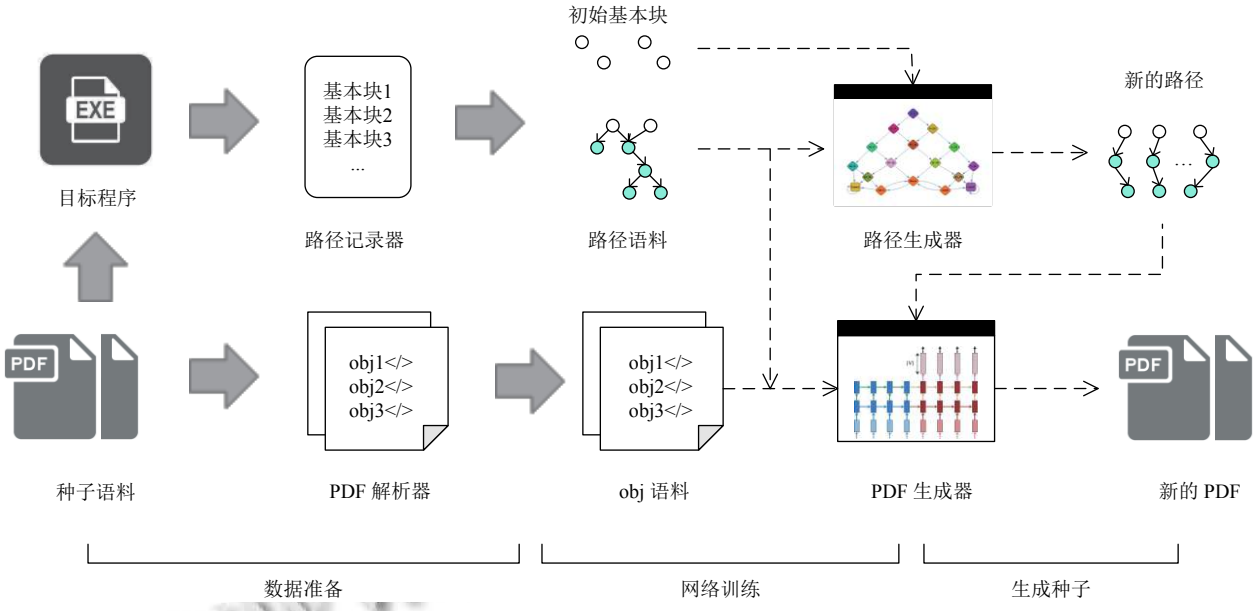

图 2 基于机器学习的框架, 通过学习语料生成新的用于模糊测试的种子

(1) 我们的框架以目标程序和种子语料 (语料指的是数据集合, 这里特指 PDF 文件集合) 作为最开始的输入. 路径记录器是用的英特尔的插装工具 Pin. 路径记录器以种子语料作为输入, 然后记录这些 PDF 在 PDF 阅读器 (目标程序) 上相应的执行路径, 生成路径语料. 这里记录的执行路径是基本块的序列. 需要说明的是, 对于第三方函数调用的基本块, 我们进行了过滤, 因为这些基本块与我们要测试的 PDF 阅读器无关, 而且第三方函数调用的基本块数目巨大, 去除掉这些无关的基本块可以提高处理基本块序列的效率. 具体的过滤方法是通过在 Pin 工具中指定目标程序, 生成的基本块序列就可以过滤掉无关的基本块, 然后再按照先后顺序排列出所有基本块就组成了相应的执行路径. 生成好的路径语料后续将作为路径生成器和 PDF 生成器的训练数据. 同时提取出的那些初始基本块序列则作为训练好的路径生成器的输入, 以生成新的路径. (这些起始的基本块序列是从路径语料中截取的基本块序列的头, 比如说一条完整的路径是<block1, block2, block3…blockn>(block 代表基本块), 而截取的路径片段可以是<block1, block2>或者<block1, block2, block3>等等).

(2) 我们把第 (1) 步里生成的执行路径中的基本块当成“字符串”来处理, 即把每条执行路径看成是基本块组成的字符串序列. 然后我们利用基于马尔可夫链的“路径生成器”来统计这些执行路径, 最后预测生成新的路径. 具体而言, 路径生成器通过给予的路径语料, 统计路径里不同的基本块之间的转移概率. 在路径生成器统计完成之后, 我们过滤出值比较小的转移概率 (为了最后生成新的基本块序列), 再给予一些起始的基本块序列片段, 路径生成器就能够预测出新的基本块序列 (即完整的新路径). 最关键的是, 生成的新路径包含了新的基本块序列 (在原始的路径语料中是没有的). 生成的新路径作为 PDF 生成器的输入, 用来指导新的 PDF 种子的生成.

(3) 最后一步对第 (2) 步生成的新基本块序列生成相应的 PDF 文件. 首先, 我们通过 PDF 解析器, 对最开始的 PDF 文件做解析, 生成相应的 obj 序列 (在第 2 节中已经说明了 obj 是 PDF 主要的文本结构). 然后我们把原始 PDF 的 obj 序列和对应的执行路径分别作为“目标”和“源”放入到 PDF 生成器 (基于 sequence-to-sequence 的神经网络) 中进行训练. 该网络通过训练可以学习到执行路径与 obj 序列之间的关系. 进而我们可以利用该网络, 以新的执行路径作为输入 (在第 2 步生成的), 输出新的 obj 序列, 最后我们把这些新的 obj 序列转化为 PDF 文件作为模糊测试的种子. 理论上, 这些 PDF 文件是拥有比较好的代码覆盖率和通过率的

高质量种子.

4 模型介绍

本节会详细地介绍框架中所用到的两个模型的原理和具体实现. 4.1 小节会介绍用于生成新路径的马尔可夫链模型, 4.2 小节会介绍用于生成新种子文件的 seq2seq 模型. 需要指出的是本文用到的神经网络模型是基于 Tensorflow 的 1.2 版本[17].

4.1 马尔可夫链模型

马尔可夫链因俄国数学家马尔科夫得名, 为状态空间中经过从一个状态到另一个状态的转化的随机过程. 具体而言, $x_1, x_2, x_3, \cdots$描述了一种状态序列, 其每个状态值取决于前面有限个状态. 如果 x_{n+1} 对于过去状态的条件概率分布仅是 x_n 的一个函数, 即:

$$P(X_{n+1}=x|X_1=x_1, X_2=x_2, \cdots X_n=x_n) = P(X_{n+1}=x_n|X_n=x_n) \quad (1)$$

则上述序列可以被称作为马尔可夫链. 马尔可夫链通常用于状态统计建模, 我们以文本生成为例, 假设我们的数据有以下四个句子:

(1) My friend makes the best raspberry pies in town.

(2) I think apple pies are the best pies.

(3) Steve thinks apple makes the best computers in the world.

(4) I own two computers and they’re not apple because I am not steve or rich.

那么我们可以统计出如图 3 所示的网络模型.

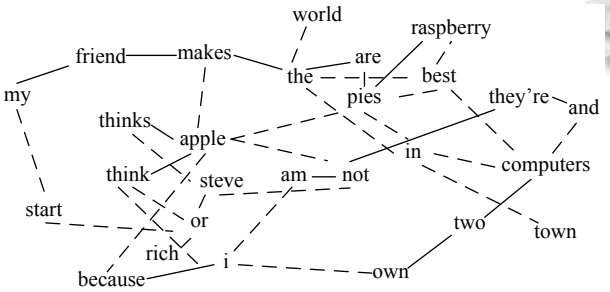

图 3 马尔可夫链网络模型

图 3 中实线代表了转移概率较大, 虚线表示概率较小. 然后我们输入起始的状态 start, 以检测到 end 作为结束, 根据转移概率就可以生成很多“新”的句子 (在原始数据集中没有出现过).

对应到我们生成基本块序列中, 我们暂且把一个基本块到另外一个基本块的转移看成是一个马尔可夫链. 我们用原始 PDF 文件集合生成的基本块序列进行统计, 得到一个概率模型. 然后我们保留概率模型中概率比较小的转移概率以便生成特殊的基本块序列. 我们以一些基本块序列的起始基本块作为输入 (前驱), 根据基本块之间的转移概率, 迭代地生成新的基本块 (后继), 直到生成 end, 迭代结束, 最后就生成了新的完整的基本块序列.

4.2 seq2seq 模型

有了新生成的路径之后, 我们需要能够执行这些新路径的 PDF 种子文件. 所以我们期望一个网络模型能够学习路径 (基本块的序列) 和 PDF 文本字符序列之间的函数关系, 这里我们用的是神经网络模型.

首先简单介绍一下神经网络的一些基本模型.

(1) CNN: 一种前馈神经网络, 由一个或者多个卷积层和顶端的全连通层组成, 同时包括关联权重和池化层. CNN 的结构能利用输入数据的二维结构, 对于图像的处理比较有效.

(2) RNN: 全称递归神经网络, 其根本特征是神经元之间既有内部的反馈连接又有前馈连接, 因此 RNN 可以利用网络内部的记忆来处理序列. 手写识别是最早成功利用 RNN 的研究成果.

(3) GAN: 全称生成对抗网络, 是非监督式学习的一种方法, 通过让两个神经网络相互博弈的方式进行学习. 该模型常常用于生成以假乱真的图片.

因为我们需要处理的数据都是序列数据, 而 RNN 可以利用网络内部的记忆来处理序列, 符合我们的应用场景, 所以我们最终选择 RNN 模型. 按照输入和输出序列的不同数量, RNN 又分为多种类型: 一对一、一对多、多对一以及多对多等. 我们的应用场景是以执行路径的基本块序列作为输入, 以 PDF 的 obj 对象序列作为输出, 是一个多对多的关系, 所以我们选择使用 seq2seq 模型, 一种可以接受变长序列的多对多 RNN 模型.

seq2seq 模型于 2014 年被谷歌提出[18], 该模型在机器翻译中的使用效果非常好. seq2seq 网络是由两个 RNN 模型组成的, 其中一个称为编码器, 另一个称为解码器. 编码器以 $X=x_1, \cdots, x_T$ 作为输入, 将其编码成一个固定长度的中间向量 c. 式 (2) 和式 (3) 说明了编码器中的原理.

$$h_t = f(h_{t-1}, x_t) \quad (2)$$

$$c = \Phi(\{h_1, \cdots, h_t\}) \quad (3)$$

解码器按照式 (4) 和式 (5) 把中间向量 c 解析成输出 y_1，y_2，…，y_T. 其中 s_t 是解码器中在时刻 t 时的隐层状态, 输出 y_t 是一个条件概率函数 P, g 是激活函数.

$$s_t = f(s_{t-1}, y_{t-1}, c) \quad (4)$$

$$P(y_t|y_{t-1}, y_{t-2}, \cdots, c) = g(h_t, y_{t-1}, c) \quad (5)$$

PDF 生成器中的 seq2seq 模型用两部分数据来训练Φ和 P 两个函数, 一部分是 PDF 的执行路径 (路径记录器所记录的基本块序列), 另外一部分是 PDF 的 obj 语料. 这里的 obj 语料是经过 PDF 解析器解析生成的每个 PDF 文件的 obj 序列. 通过足够的训练之后, PDF 生成器就能够拟合出基本块序列和对应 PDF 的 obj 序列之间的函数关系. 然后我们把利用马尔可夫链生成新的基本块序列输入到 PDF 生成器, 就能够输出对应的新的 obj 序列, 最后只需要把这些 obj 对象进行连接, 并加上合适的文件头和文件尾, 就完成了最终的 PDF 文本的生成.

5 实验评估

我们进行了一系列的实验来评估我们的系统框架在提高种子 (PDF 文件) 质量方面的有效性. 我们的实验主要回答了以下 3 个问题:

(1) 我们的框架能否生成质量更高的种子文件, 即我们生成的种子文件是不是能够触发更多新的基本块.

(2) 我们的框架是不是能生成有效的 PDF 文件从而通过阅读器的语法语义检查 (只有通过语法语义检查才能进行更深入的代码检测).

(3) 我们的框架针对某个 PDF 阅读器生成的 PDF 文件对于其他阅读器进行模糊测试, 其代码覆盖率能否也有所提高.

5.1 实验环境

我们主要的目标软件是 CAJViewer 阅读器 (7.2 版本). 我们在网上爬取了 43 684 个 PDF 文件 (大小为 4.4 GB), 作为最原始的种子文件. 这些种子文件由 PDF 解析器解析成 obj 序列给 seq2seq 网络使用. 同时, 路径记录器把这些原始的 PDF 送入到 CAJ 阅读器中, 并记录每个 PDF 文件的基本块序列.

马尔可夫链的统计和预测以及 seq2seq 网络的训练和预测在 16.04 版本的 Ubuntu 系统下进行, 该系统拥有 4 核的 i7-7700 处理器, 16 GB 的 RAM, 和 NVIDIA GTX1080TI GPU. 两个模型的统计/训练时间为 24 小时. 路径记录器和 PDF 解析器则运行在 32 位的 Windows7 虚拟机下 (使用虚拟机是因为使用的路径记录器 Pin 工具只能在虚拟机环境下才可以正常获取 PDF 文件在 CAJ 阅读器中的执行路径).

基于马尔可夫链的概率模型统计完成之后, 我们把初始的基本块序列片段输入到模型中, 生成新的基本块序列, 然后将其输入到训练好的 seq2seq 网络中, 最后生成新的 PDF 文件加上原始的 PDF 文件作为模糊测试的种子.

5.2 代码覆盖率

我们进行了两组实验来评估种子文件的代码覆盖率. 第一组实验统计新生成的 PDF 文件覆盖了多少新的基本块. 第二组实验统计新生成的路径 (只要有新的基本块或者不同的基本块顺序组合, 就认为是新的路径), 这里我们之所以考虑新的生成路径是因为对于模糊测试而言, 新的路径也代表了文件在程序中新的行为, 这对于模糊测试而言也是有帮助的.

5.2.1 新的基本块

首先我们把原始的 43 684 个样本文件进行过滤, 去除掉执行路径相同的样本之后剩下 14 522 个 PDF 文件. 然后, 我们把剩下的文件分组成四组, 每组包含了 3630 个文件. 我们设定一个基准测试集: B1、B2、B3、B4 分别包含了第一组 PDF, 前两组、前三组和全部的 PDFs.

同时, 我们用训练好的网络分别对 B1-B4 中的文件 (作为训练数据) 生成新的 PDF, 最后新生成的 PDF 组成了 T1、T2、T3、T4 四组数据. 我们把 T1–T4 和 B1–B4 共 8 组数据分别放入到目标程序中执行, 统计出他们覆盖的基本块数量, 并计算出生成的新的基本块数目, 如图 4 所示.

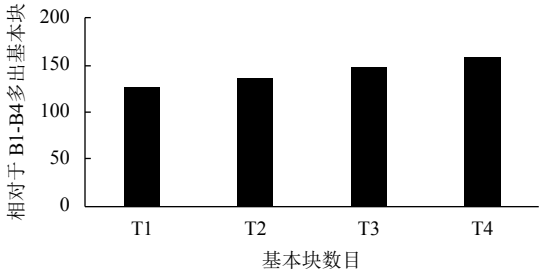

图 4 新生成的基本块数目

从图中我们可以看到, 随着给目标程序输入越多的 PDF 文件, 新生成的基本块也越来越多, 最终对比 T4 和 B4 可以看出新生成的基本块一共有 159 个. 需要指出的是, 这里新生成的基本块指的是 T4 生成的基本块去除 B4 生成的基本块中相同的基本块之后剩下的集合. 经过我们的统计, 原始的 43 684 个 PDF 一共覆盖了 6403 个基本块, 占目标程序所有基本块 (45 991 个) 中的 13.92%. 所以生成的新 PDF 文件覆盖到的新基本块 (159 个) 有 0.35% 的提升. 而且原始 PDF 没有触发目标程序任何 crash, 而生成的新 pdf 文件一共触发了 338 个 crash.

以下是和其他相关工作的对比. Rajpal 等人[16]利用 RNN 网络来对 AFL 进行改进, 提高种子筛选的策略. 他们以 mupdf(一款开源的 PDF 阅读器) 为研究对象, 他们用改进后的 AFL 使得模糊测试的覆盖率提高了 0.17%. 他们也对其他文件格式做了测试, 实验表明他们的方法对于 PNG 等比较简单的文件格式的效果要更显著一些. Patrice 等人[18]使用 RNN 网络学习 PDF 的文件格式以生产新的 PDF 文件, 他们是以指令为指标统计代码的覆盖率. 他们的方法能够覆盖到 6658 条新的指令, 相对于目标程序中 560 K 条指令, 提高了 0.11% 的覆盖率. 而我们的方法提高了 0.35% 的代码覆盖率, 比上述两个工作的实验结果都要好. 另外 Skyfire[17]生成的种子文件可以提高 20% 的覆盖率, 但是他们的实验对象是一些脚本语言, 相比 PDF 的文件结构, 他们的模型所要学习的语法更为简单.

5.2.2 新的路径

除了新的基本块, 我们还专门统计了新生成的 PDF 相比原始的 PDF 所触发的新路径. 这里我们遇到的一个问题是 CAJViewer 再打开 PDF 文件之后不会立即关闭, 而是在那边循环等待. 因此, 路径记录器所记录的路径末端都包含了很多无用的循环路径, 所以为了截取有效的执行路径, 我们找到了一个特定的基本块 (保证该基本块之后 CAJViewer 就开始循环等待), 把该基本块之前的基本块序列截取出来. 然后检验这些截取后的基本块序列是不是新的路径. 实验结果如图 5 所示.

从图中我们可以看到随着生成的 PDF 数目的增加, 触发的新路径数目也在稳步增加, 最后达到了 1008 条. 实验结果表明, 我们生成的新 PDF 不仅可以触发新的基本块, 还可以触发一定数量的新路径.

5.3 文件通过率

我们做的第二个实验是评估我们的框架能否生成有效的 PDF 文件来通过阅读器的语法语义检查. 我们把 5.2.1 节中生成好的四组 PDF, T1、T2、T3 和 T4 输入到 LibreOffice 提供的一款命令行工具中. 该工具会把 PDF 文件解析成 txt 文件, 然后返回一个值用来表示是否成功解析了 PDF 文件. 我们用该工具统计生成的 T1–T4 四组 PDF 文件的通过率.

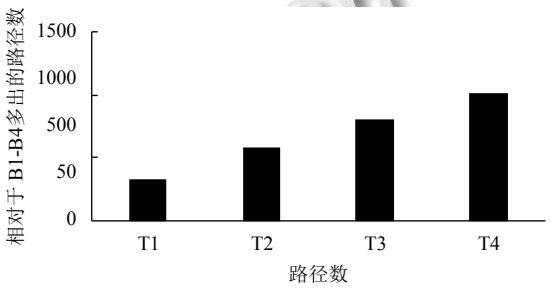

图 5 新生成的路径数目

我们以 AFL 生成的 PDF 文件作为 Baseline. 即我们把 5.2.1 节中的 B1-B4 四组 PDF 放到 AFL 中做模糊测试, 生成和 T1-T4 一样数量的 PDF 文件, 然后对比两者的通过率. 实验结果如图 6 所示.

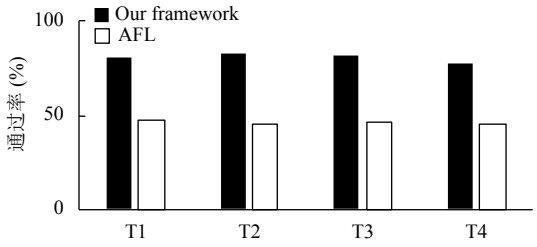

图 6 我们的框架和 AFL 所生成的 PDF 各自的通过率

从图中我们可以看到, 我们的方法生成的 PDF 有着较高的通过率 (大概在 80% 左右), 远高于 AFL (40% 左右). 但是需要说明的是, 模糊测试本身并不要求种子的通过率越高越好 (因为要保证生成的种子越特殊, 其通过率可能就会越低). 但是种子最好有合适的通过率以便保证模糊测试的效率. 尽管我们生成的 PDF 文件的通过率可以通过提高模型的训练时间而变得更高, 但是我们认为 80% 的通过率已经足够.

5.4 普遍适用性

我们的路径记录器是基于 CAJViewer 为目标程序去统计的执行路径, 所以我们的网络模型是针对

CAJViewer 训练的. 所以我们想要评估该模型是否适用于其他的 PDF 阅读器, 即我们针对 CAJViewer 生成的种子文件对于其他的 PDF 阅读器的代码覆盖率是不是也有相应的提高.

我们的第三个实验把原始的 PDF 集合和生成的新 PDF 集合输入到 3 个轻量级的 PDF 阅读器中, Smart PDF, Sumatra PDF 和 Jisu PDF(没有用 Adobe 或者 Foxit Reader 作为测试对象是因为这两款软件不方便使用工具来统计覆盖率) 来统计生成的新 PDF 所提高的代码覆盖率. 具体结果如表 1 所示.

表 1 针对不同的 PDF 阅读器统计的覆盖率

结果内容	Smart PDF	Sumatra PDF	Jisu PDF
原始的 PDF 覆盖的基本块	2 902 384	2 403 803	5 216 173
生成的 PDF 覆盖的基本块	2 340 198	1 838 743	3 366 689
生成的 PDF 覆盖的新基本块	11 222	16 154	133 114
生成的 PDF 提高的覆盖率 (%)	0.38	0.67	2.55

从表 1 中可以看到, 虽然生成的 PDF 所覆盖的基本块总数没有原始的 PDF 覆盖的多, 但是他们仍然覆盖了很多原始 PDF 所没有覆盖到的新基本块. 可以看到针对这三款 PDF 阅读器所提高的覆盖率都比 CAJViewer 的 0.35% 要高. 特别是 Jisu PDF, 提高了将近 2.55% 的代码覆盖率. 从这些实验结果可以看出我们针对 CAJViewer 阅读器训练的网络模型对于其他阅读器的代码覆盖率的提高也有不错的效果, 即说明了我们的框架有着不错的普适性.

6 讨论

实验表明, 我们的框架有效地提高了初始种子输入的代码覆盖率, 但是, 我们的工作仍然有一些可以改进的地方.

首先, 我们生成的 PDF 质量不够完美, 这可以从实验结果中看出, 生成的 PDF 覆盖的基本块总数比原先的 PDF 要少. 导致生成的 PDF 质量不够高的原因可能是 PDF 生成器没有训练完全, 没有充分学习到 PDF 的语法结构, 这个问题需要后期去完善网络模型.

其次, 我们使用的神经网络模型主要是基于 LSTM 单元的, 而目前有很多 LSTM 的变形以及关于 RNN 的很多新的模型. 可以把这些新的模型以及变形作为训练网络, 进行对比实验, 来评估哪种模型更加适合做 PDF 文件的生成. 该工作可以作为未来的一项研究内容.

最后, 我们的框架应该同样能够针对其他的文本文件做种子生成. 即如果机器学习可以提高针对 PDF 这种复杂结构的种子的质量, 那么应该也可以用于其他更加简单的文本结构. 在文献[16]的工作中, 作者也有上述类似的观点. 而且用他们的方法针对简单的文本格式 (比如 ELF、PNG 文件) 比针对 PDF 这种复杂文本的模糊测试效果要好. 类似地, 我们也需要评估我们的框架对于简单的文本格式的模糊测试效果, 这也可以作为未来的工作.

7 总结

本文提出了一种系统框架, 利用机器学习来挖掘 PDF 文件和其在目标程序中执行路径的关系, 并且利用这种关系去生成新的 PDF 文件来提高模糊测试的代码覆盖率. 实验结果表明, 对比相关工作, 我们的框架生成的 PDF 文件对代码覆盖率有相对较高的提升 (0.35%–2.55%), 而且我们的方法同时保证了生成的 PDF 文件有良好的通过率. 就覆盖率和通过率上而言, 我们的方法可以生成质量较高的种子文件, 这些种子文件将提高后续模糊测试的效果.

参考文献

1 杨梅芳,霍玮,邹燕燕,等. 可编程模糊测试技术. 软件学报, 2018, 29(5): 1258–1274. [doi: 10.13328/j.cnki.jos.005499]

2 Wang JJ, Chen BH, Wei L, *et al*. Skyfire: Data-driven seed generation for fuzzing. Proceedings of 2017 IEEE Symposium on Security and Privacy (SP). San Jose, CA, USA. 2017. 579–594.

3 Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. Urbana, IL, USA. 2017. 50–59.

4 Godefroid P, Levin MY, Molnar D. SAGE: Whitebox fuzzing for security testing. Queue, 2012, 10(1): 20. [doi: 10.1145/2090147]

5 Haller I, Slowinska A, Neugschwandtner M, *et al*. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. Proceedings of the 22nd USENIX Conference on Security. Berkeley, CA, USA. 2013. 49–64.

6 Pak BS. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution[Master's thesis]. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 2012.

7 Stephens N, Grosen J, Salls C, *et al*. Driller: Augmenting fuzzing through selective symbolic execution. Proceedings of the 23rd Annual Network and Distributed System Security Symposium. San Diego, CA, USA. 2016. 1–16.

8 Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. Proceedings of the 31st International Conference on Software Engineering. Vancouver, BC, Canada. 2009. 474–484.

9 Li YK, Chen BH, Chandramohan M, *et al*. Steelix: Program-state based binary fuzzing. Proceedings of the 11th Joint Meeting on Foundations of Software Engineering. New York, NY, USA. 2017. 627–637.

10 Pham VT, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Singapore. 2016. 543–553.

11 Rawat S, Jain V, Kumar A, *et al*. VUzzer: Application-aware evolutionary fuzzing. Proceedings of the 24th Annual Network and Distributed Systems Security. San Diego, CA, USA. 2017.

12 Böhme M, Pham VT, Nguyen MD, *et al*. Directed greybox fuzzing. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, TX, USA. 2017. 2329–2344.

13 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria. 2016. 1032–1043.

14 Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. Proceedings of 2015 IEEE Symposium on Security and Privacy. San Jose, CA, USA. 2015. 725–741.

15 Woo M, Cha SK, Gottlieb S, *et al*. Scheduling black-box mutational fuzzing. Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. Berlin, Germany. 2013. 511–522.

16 Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv: 1711.04596, 2017.

17 Abadi M, Barham P, Chen JM, *et al*. TensorFlow: A system for large-scale machine learning. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah, GA, USA. 2016. 265–283.

18 Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. arXiv: 1409.3215, 2014.