计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(6):1−8 [doi: 10.15888/j.cnki.csa.007929]

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

基于机器学习的模糊测试种子输入优化①

王　敏[1,2], 冯登国[1,2], 程　亮[2], 张　阳[2]

[1](中国科学技术大学, 合肥 230026)
[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)
通讯作者: 王　敏, E-mail: kdyx014@mail.ustc.edu.cn

摘　要: 模糊测试作为一种自动化检测应用程序漏洞的方法, 常常被用来检测各种软件以及计算机系统的漏洞挖掘中. 而种子文件质量的高低对于模糊测试的效果而言至关重要. 所以本文提出了一种基于机器学习的模糊测试种子输入的生成方法, 利用样本输入和基于机器学习的技术来学习样本输入的规则和语法. 并利用学到的规则和语法来生成全新的种子输入. 我们还提出了一个采样方法. 使得这些新的种子输入的覆盖率较之前有了明显提升.

关键词: 模糊测试; 机器学习; 代码覆盖率; 种子生成; Transformer 模型; attention 机制

引用格式: 王敏,冯登国,程亮,张阳.基于机器学习的模糊测试种子输入优化.计算机系统应用,2021,30(6):1–8. http://www.c-s-a.org.cn/1003-3254/7929.html

Optimization of Fuzzing Seed Input Based on Machine Learning

WANG Min[1,2], FENG Deng-Guo[1,2], CHENG Liang[2], ZHANG Yang[2]

[1](University of Science and Technology of China, Hefei 230026, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: As a method of automatically detecting application vulnerabilities, fuzzing often serves for various software and computer systems. The quality of the seed file is very important to the fuzzing test. Therefore, this study proposes a method for generating fuzzing seed input based on machine learning. It relies on sample input and machine learning-based technology to learn the rules and grammar of sample input, which are then used to generate new seed input. We also propose a sampling method, considerably improving the coverage of the new seed input.

Key words: fuzzing; machine learning; code coverage; seed generation; Transformer model; attention mechanism

1　引言

随着计算机技术的迅猛发展以及在各个行业中日益广泛的应用. 人们的生活和生产方式发生了前所未有的改变, 计算机在人们生活、工作生产、国防事业、科技研究中所起到的作用越来越无法代替. 各行各业都在依靠计算机来实现生产、管理、销售、控制等各个主要流程. 然而伴随而来的计算机安全性的问题也在日益陡增. 软件作为计算机中, 不可或缺的一部分, 软件的安全性也受到了十分严峻的考验. 软件漏洞, 又称为 Bug, 指的是软件在具体实现或系统安全策略上存在的缺陷. 攻击者可以利用漏洞来完成未授权的访问或者是对系统的破坏. 一些我们耳熟能详的蠕虫病毒、木马病毒, 都是利用软件漏洞来植入的, 它们会窃取受害者的信息, 拒绝服务攻击, 传染给互联网上其他的计算机. 某些更严重的漏洞可以被用来执行黑客想执行的任意代码. 漏洞的危害不言而喻, 所以想要避

① 基金项目: 国家自然科学基金 (61471344, 61772506, 62072448); 国家重点研发计划 (2017YFB0802902)
Foundation item: National Natural Science Foundation of China (61471344, 61772506, 62072448); National Key Research and Development Program of China (2017YFB0802902)
收稿时间: 2020-10-09; 修改时间: 2020-11-02; 采用时间: 2020-11-04; csa 在线出版时间: 2021-06-01

免这些危害一个直接的办法便是在这些漏洞被攻击者加以利用之前首先找到并修复这些安全漏洞.

然而, 程序漏洞的数量非常庞大. 例如 Image Tragick 这款软件, 每个月都可以在这款软件中找到新的漏洞, 并且每年都会发现一些影响比较大的严重漏洞, 2017 年就发现了 357 个 CVE. 对于如此大的危害, 我们急需一些方法和手段, 在黑客利用这些漏洞之前找出它们. 找出安全漏洞的技术被称为漏洞挖掘. 在目前漏洞挖掘领域中, 有一个越来越被广泛使用的方法: 模糊测试[1], 它的基本流程如图 1 所示.

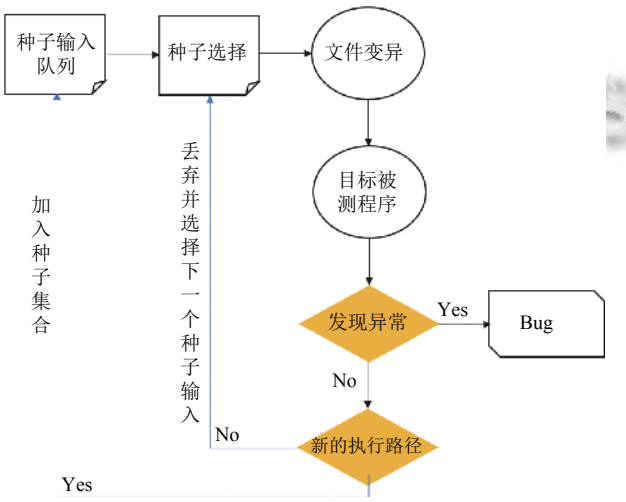


图 1 模糊测试基本流程图

模糊测试通过某些策略, 随机生成大量的非预期输入种子文件, 以这些种子作为目标程序的输入, 观察目标程序是否会出现执行异常或者崩溃, 以此来检测出目标程序中的漏洞. 模糊测试相当于是去解一个搜索问题, 即从一个预先构造好的种子输入开始, 搜索经过几百万次变异之后, 覆盖率最高, 崩溃最多的种子集合. 由于其自动化程度较高, 所以得到了安全人员的大量广泛的使用. 模糊测试目前有很多种流派, 从编译策略的不同选择可以大致分为 3 种: 白盒模糊测试[2]、黑盒模糊测试[3]、灰盒模糊测试[4]. 白盒模糊测试指的是对目标程序的源代码和程序内部的逻辑结构完全了解的情况下, 针对性地对程序进行分析后执行模糊测试, 以执行程序中更多的代码块, 提高代码覆盖率; 黑盒模糊测试则是无法获取目标程序的源代码, 对目标程序内部的代码逻辑一无所知. 大部分情况下, 源代码等都是公司或者机构的机密, 无法轻易获取. 所以针对这种黑盒情况下一般只会采用一些简单的随机变异的方法后进行模糊测试. 灰盒模糊测试介于白盒模糊测试与黑盒模糊测试之间. 灰盒模糊测试会同时考虑代码程序的逻辑结构以及观察目标程序执行时的输出来获取一些有价值的模糊测试信息, 利用这些信息来更好地指导模糊测试的效果提升. 然而现有的灰盒模糊测试工具的效果不尽如人意, 根据文献 [5] 中的实验结果, 拿目前最流行的灰盒模糊测试工具 AFL (American Fuzzy Lop) 来说, 它的测试通过率也仅仅不到 30%, 所以灰盒模糊测试的代码覆盖率仍有足够的提升空间. 为了解决当前模糊测试测试率低且代码覆盖率不高的问题, 我们提出了一个基于机器学习的框架来提高模糊测试的有效性和效率. 该框架用来发现目标种子输入和待测目标程序执行之间的相关性, 然后用该相关性继续指导新种子输入的生成. 然后利用新生成的种子文件作为模糊测试的输入, 研究新生成的种子输入文件对目标程序的代码覆盖率的影响, 从而达到探索目标待测试程序更多行为的目的.

本文的主要贡献如下:

(1) 我们提出了一个基于机器学习的框架来生成模糊测试的种子文件, 利用 Transformer 模型来学习 PDF 文件内部的格式化文件语法, 并指导生成全新的完整 PDF 文件, 用于后续的模糊测试. 实验表明对于 mupdf 这款 PDF 阅读器, 我们的代码覆盖率有了一定提高.

(2) 我们提出了两种采样算法来对学习的分布进行采样, 在确保 obj 对象序列依据概率分布进行预测的同时, 保证了生成结果的多样性. 大大减少了生成的 obj 对象序列存在较多重复这一问题.

文章后续章节的内容如下: 第 2 节阐述了相关研究与背景, 第 3 节详细介绍了我们设计的框架细节, 第 4 节阐述了本文所使用的模型以及在模型上所做的适应性改进. 第 5 节介绍了我们实验的结果与评估. 最后第 6 节是总结和展望.

2 相关研究与背景

2.1 相关工作

目前国内外对于提高模糊测试的效率有很多工作, 这些工作大致可以分为两种策略: 一种是改进其变异机制来智能地生成更可能触发目标程序中潜在错误和崩溃的测试输入来实现增强模糊器的功能; 另一种是提高种子输入的质量, 以便能够探索目标程序的更多

代码和行为.

改变变异机制是提高模糊测试代码覆盖率的一种方案, 大致可以把改变变异机制的工作分为两类: 第 1 种基于程序的方法关注程序本身, 第 2 种基于模糊测试的方法侧重于模糊测试本身. 第 1 种方法会利用一些程序分析的方法来探索目标程序和种子输入文件之间的联系. 例如, 文献 [6] 中利用了符号执行的技术提出了一种混合测试方法, 该方法将符号执行技术和模糊测试结合起来. 符号执行是一种常用的程序分析技术, 它不向程序提供正常的输入 (如数字), 而是提供代表任意值的符号.除了值可以在输入符号上使用符号公式外, 执行过程与正常执行一样进行. 符号执行理论上是对路径约束求解, 所以能够发现和探索程序中所有可能的路径, 但实际上是不可扩展的, 因为路径的数量很快会呈指数级增长, 实际情况下不可能提供无限大的资源, 因此符号执行的缺点是费时费力. 而模糊测试比符号执行快得多, 一台机器就可以完成一项模糊测试, 因此模糊测试可以更深入地探索代码, 但是它在广度上限制了代码覆盖的范围. 所以混合测试是利用符号执行来扩展到各种不同的 (唯一的) 路径, 然后使用模糊测试来快速测试每个路径, 从而更好地提高代码覆盖率.

第 2 种方法是基于模糊测试的, 该方法会从已经执行过的模糊测试中学习一些有用的经验, 这些经验可以更好地指导种子变异机制, 从而变异出更高质量的种子. 例如, 文献 [7] 中提出了一种基于覆盖的灰盒模糊测试方法 (CGF), CGF 没有使用任何程序分析的技术, 它使用轻量级的二进制工具来确定由输入执行的路径的唯一标识符. 如果模糊出一个新的和有趣的路径, 模糊测试会保留那个输入, 否则, 它将丢弃该输入.

这些方法都在一定程度上提高了代码覆盖率, 但是还是有一定的不足, 例如基于程序的方法利用到的符号执行方法效率很低, 对新生成的种子输入的代码覆盖率没有显著提高. 另外更重要的一点是, 当种子文件的格式十分复杂时[8], 普通的基于变异策略的方法产生的新的种子文件很可能连最初的语法检查都没法通过, 导致生成大量的无效种子, 从而也无法明显提高代码覆盖率. 由于基于变异机制的方法, 针对于复杂格式的种子文件, 想要提升代码覆盖率是比较困难的, 因为这种方式生成的新的种子输入文件很可能无法通过语法检查器的检查. 无论变异机制有多好, 我们都相信初始种子语料库的质量, 就其在目标计划中可以探索的代码量而言, 对于模糊测试的成功具有最重要的影响. 其中 AFL 仅在初始种子语料库覆盖的 15 个月模糊测试后, 能够覆盖目标程序 (脚本解析器) 中多 5.1% 的行.

目前国内外有很多工作提出了各种方法来提高输入种子文件的质量来进行模糊测试, 这也是与我们工作最相关的方法. 文献 [9] 提出了一种从原始语料库中选择最合适的种子输入的方案, 以此来最大化目标程序的代码覆盖率. 最近机器学习的火热也让很多工作利用神经网络来进行程序分析和漏洞检测相结合, 例如, 文献 [10,11] 提出了一些神经网络来对数组进行排序和复制算法进行学习. 文献 [12] 提出了一种叫 Neural FlashFill 的方法来在特定领域的语言上生成基于正则表达式的代码. 文献 [13–15]用到了 Seq2Seq 这样先进的深度神经网络模型在正确的程序上来学习语法知识, 然后基于学到的知识来指导修复程序中的语法错误. 文献 [16] 利用机器学习方法学习种子语料库的输入种子文件的语法和语义, 并利用学习到的这种相关性来更好的指导新的种子输入的生成, 实验表明他们的方法对于代码覆盖率的提升确实起到了一定的作用. Nichols 等在文献 [17] 中提出了联合 LSTM (长短期记忆人工神经网络)[18] 网络和 GAN (生成式对抗网络)[19] 来学习整个种子语料库中的语法信息, 并生成新的种子文件. 但是他们都没有生成一个全新的格式良好的复杂种子文件例如 PDF 文件, 这导致了他们新生成的初始种子与原始种子库中的种子相比, 覆盖范围的提升很有限并且存在很大的重复.

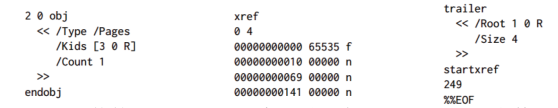
2.2 背景介绍

PDF 文件格式是现实软件应用程序可能需要处理的复杂输入的一个很好的例子, 这是模糊测试技术面临的主要挑战之一. 当前版本的 PDF 文件格式规范, 长度超过 1300 页, 将每个 PDF 文件定义为扁平位流, 包含以下 4 个部分:

(1) 标题部分, 声明 PDF 规范版本后跟文件, 该文件通常位于单行中, 例如“%PDF-1.7”.

(2) 正文部分, 由一系列表示 PDF 文件内容的对象组成. 如图 2(a) 所示, PDF 文件中的对象由两部分组成: 1) 前两个数分别是对象的标识符和当对象被修改为更新时增加 1 的世代号; 2) 由关键字“obj”和“endobj”包围的对象的内容. 对象可以采用 8 种有效类型中的任何一种, 即布尔值、数字、字符串、名称、数组、字典、流和 NULL. 布尔、数字和字符串类型与编程语言中的类型相同, 而数组和字典类型分别由

“[”/“]”和“«”/“»”对分隔. 流类型通常用于大量数据, 并由流和端流分隔. PDF 文件中的名称类型, 以“/”开头, 后跟一系列字符, 定义 PDF 文件中使用的键. 例如, 图 2(a) 中的/Type/Pages 显示该对象代表 PDF 文件中的页面; /Count 1 声明该页面只有一个子对象; /Kids [3 0 R] 指定子对象的标识符为 3, 其世代号为 0.



(a) PDF 主体的一个 obj 对象　(b) 交叉引用表　(c) PDF 文件尾

图 2　PDF 文件结构

(3) 交叉引用表, 用于快速访问 PDF 文件中的对象. 该表以关键字 xref 开头, 后跟一条指示起始对象的行 (例如, 图 2(b) 中第 2 行的 0) 和文件中的对象数 (例如, 图 2(b) 中第 2 行的 4). 表的其余部分是许多条目, 为 PDF 文件中的每个对象提供参考信息. 每个条目都是一行声明, 其中前 10 个数字表示对象的地址 (作为文件中的偏移量), 后面的 5 个数字定义对象的世代号, 结束字符表示对象是否为占位符 (表示为 f) 或使用中 (表示为 n).

(4) 文件尾部分, 定义 PDF 文件的其他元数据, 例如每个部分的地址和大小. 文件尾部分包含一个至少有两个条目的字典: /Root 指的是 PDF 文件 1 中的第一个对象; 和/Size 表示交叉引用表中的条目数. 关键字 startxref 和下一行中的数字表示交叉引用表的起始地址 (即关键字 xref 的文件中的偏移量).

3　框架概述

我们的系统框架如图 3 所示. 我们将种子语料库中的 PDF 的 obj 对象提取出来, 生成包含很多 obj 对象的 obj 对象库, 然后经过 obj 对象生成器, 生成新的 obj 对象. 再结合 PDF 的结构, 在 PDF 组装器中生成完整的新的 PDF 种子. 最后便是将生成的全新种子投入到模糊测试中, 观察代码覆盖率是否提升和种子质量的好坏. 我们的框架主要分为 3 个部分:

第 1 部分为框架的输入. 输入为 obj 对象, 我们将原始语料库中的所有 PDF 种子文件的 obj 对象抽取出来, 这里的 PDF 种子文件都是经过错误检测和数据清洗的, 我们去掉了格式错误以及重复的 PDF 种子文件, 确保它们的正确性. 然后如如图 2 所示. 我们截取“20 obj”和“obj”中间的这段主体部分. 之后对 obj 对象做一些格式上的处理. 首先我们只选择主体长度在 50 以内的 obj 对象, 因为长度过长的 obj 对象在学习时会因为依赖性的问题导致学习效果的下降. 其次我们在标签之间用<ENT>做特殊标记, 表示换行符, 方便将 obj 展开成一个字符序列, 同时也方便之后我们生成的 obj 序列翻译回格式正确的 obj 对象. 最后, 对于 PDF 中的二进制对象, 我们用<stream>做特殊标记, 由于二进制数据量非常大, 并且针对二进制的全自动黑盒和白盒模糊测试已经被证明有效, 我们只是将它作一个特殊标记, 不去具体学习它的内在内容, 并且也方便之后生成的 obj 序列翻译回格式完整的 obj 对象.

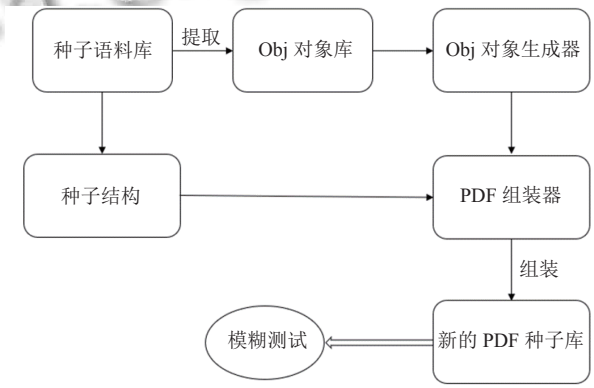


图 3　基于机器学习的模糊测试框架图

第 2 部分是 obj 对象生成器, 它是由 Transformer 网络构成. 我们将第一步生成的 obj 对象序列当做一个由若干个单词所组成的句子来处理, 即我们把每个 obj 序列看成是由一个一个标签或者符号组成的字符串序列. 然后我们将这些序列输入到基于 Transformer 网络的 obj 对象生成器中来生成新的 obj 对象序列. 我们通过将输入序列右移位一个位置得到对应的输出序列. 具体而言, obj 对象生成器会学习第一步得到的 obj 序列语料, 学习标签和符号之间的关系和转移概率. 我们根据这种学习到的知识, 加上给定的起始 obj 序列的片段, 就可以指导性地预测出新的 obj 对象序列. 并且, 我们也在这其中人为地引入了两种随机采样的方案, 我们称之为 Sample 采样和 SampleFunc 采样, 在保证生成的 obj 对象序列格式正确率的同时, 增加一些随机采样的方法, 来生成各种新的 obj 对象序列.

第 3 部分是 PDF 组装器. 我们将由 obj 对象生成的新的 obj 对象序列经过 PDF 组装器得到新生成的 PDF 文件, 作为模糊测试的种子输入. 我们从种子语料库中随机选择若干个 PDF 文件作为初始种子, 并获取

他们的结构. 然后我们利用这些 PDF 的结构, 结合新生成的 obj 对象, 组装成一个完整的 PDF. 具体而言, 我们将新生成的 obj 对象中的<ENT>特殊标记转化为空格, 将<stream>特殊标记转化为原始种子语料库中的随机一个二进制流文件. 最后我们将 PDF 文件的文件头、文件尾以及交叉引用表生成, 并将几个部分组装在一起, 形成最后格式完整的新的 PDF 种子文件.

4 模型介绍

本节将详细地介绍框架中所用到的神经网络模型和它的具体实现. 本文所用到的 Transformer 模型是基于 Python3.6.8 + PyTorch1.5.0 实现的.

4.1 Transformer 模型

Transformer 模型由 Google 在 2017 年 6 月发表的文章《Attention is all you need》提出[20]. 自从 attention 机制问世以来, 就已经成为了 Seq2Seq 网络的标准配置. 然而传统的 Seq2Seq 网络还是需要用 CNN 或者 RNN 等网络来作为网络架构的主体. Transformer 网络创新性的摒弃 Seq2Seq 网络必须结合 CNN 或者 RNN 网络的固有模式, 只采用了 attention 机制来减少网络参数的计算量, 以及解决传统 Seq2Seq 网络的并行计算效率低的弊端. 并且 Transformer 网络在绝大多数自然语言处理领域的任务上的表现, 都有一定的提升. Transformer 模型其本质上是一个 encoder-decoder 的结构.

4.1.1 Encoder

Transformer 的 encoder 是由多层堆叠而成. 一般而言层数为 6. 其中, 每层又可以分为两个子层, 分别是 multi-head self-attention 子层和 position-wise feed-forward networks 子层. 每个子层都经过了残差连接和归一化, 所以每个子层的输出为:

$$sub\_layer\_output = LayerNorm(x + (SubLayer(x))) \quad (1)$$

Transformer 网络引入了 self-attention 机制, multi-head attention 即定义多组的 Q, K, V, 让注意力 head 分别关注到不同的上下文信息, 然后将这些注意力的结果拼接在一起:

$$MultiHead(Q,K,V) = Concat(head_1, head_2 \cdots, head_h) W^O \quad (2)$$

其中, 每个$head$的计算方法为:

$$head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right) \quad (3)$$

针对 PDF 这种复杂格式的文件, 我们将 PDF 的主体部分拆分为若干个如图 2(a) 所示的 obj 对象. 在 encoder 部分, 我们将 obj 对象进行序列化, 以匹配 Transformer 模型的输入格式. 具体做法为: 对于 obj 对象, 我们总是以 obj 作为 obj 序列的开头, 以 endobj 作为序列的结尾. 我们将这些标签或者关键词看做是一个一个单词, 那么整个 obj 对象就可以看成是一个句子. 我们在句子的开头和结尾加上 BOS 和 EOS 特殊标记来标志句子的开头和结尾, 这样我们的 obj 对象就可以作为 Transformer 网络的合法输入进行正常的训练和预测了.

4.1.2 Decoder

Decoder 的输入是 encoder 层的输出和前一个位置的 decoder 的输出. 所以中间的 attention 的 K, V 是来自于 encoder 层. Q 则是来自前一个位置的输出. Decoder 的输出是对应位置的输出词的概率分布. Decoder 层的结构与 encoder 层的结构大致差不多, 只是多了一个 masked multi-head attention 层. 这里的 mask 指的是对未来的数据进行 mask 遮蔽, 防止解码预测该词时已经看到未来的词语从而导致作弊.

在 decoder 的部分, 我们对 obj 对象做与 encoder 部分中类似的处理. 不同的是由于我们需要将下一个位置的单词作为我们当前词的预测输出, 所以在 decoder 中我们 obj 序列的所有单词都会向右移动一个单位长度来指导 Transformer 输出下一个单词. 还有一个特别的操作在于, decoder 部分在 masked multi-head attention 层有一个 mask 操作, 这个操作会对未来的词语进行遮蔽防止作弊.

4.2 采样策略

对于 Transformer 模型而言, 预测下一个字符的策略是选择最大概率的字符, 在本实验中即预测 obj 对象序列时, 下一个标签或者特殊符号的预测值总为概率最大的那个值. 但是这不适合我们的 obj 对象序列生成, 因为一旦模型训练完毕, 各字符之间的概率分布就已经确定了, 而本文的 obj 序列生成策略是给定 obj 序列的开头为字符串“obj”, 让 Transformer 模型不断贪心的预测下一个字符直到序列的句尾为止, 所以结合模型的特性, 一旦 obj 序列的开头给定了, 那么模型总是会生成相同的 obj 序列, 这样造成的后果是模型生成的 obj 序列都是重复的, 并且由序列组成的 PDF 种子文件都是也都是重复的. 虽然这样的策略生成的 obj 序列总是符合语法的, 但是对于模糊测试而言, 重复的合法种子无法给模糊测试带来任何有效贡献.

因此, 我们设计了两种不同的采样策略来对 obj 对象序列生成过程中学习到的条件分布进行采样, 以确

保生成的 obj 对象序列的多样性:

Sample: 每次生成下一个字符时, 我们都会对学习的分布进行采样. 这确保了 obj 对象序列生成的多样性, 但是过多的随机采样可能会增加生成不符合 PDF 格式检查的 obj 序列的风险.

SampleFunction: 仅当我们预测的下一个字符是<ENT>时, 我们才对学习的分布进行采样. 因为遇到<ENT>时代表了我们的 obj 对象序列当前行已经预测完毕. 我们保留了整行的完整性, 只有在下一行时才进行采样. 这样减少了生成的 obj 对象序列无效性的风险.

两种采样方案都在模型预测之前, 对预测的结果进行采样, 这样做从一定程度上破坏了网络学习到的语法规则, 因为最大概率的输出是最符合语法规则的, 而采样之后输出的是概率比较小的预测字符, 这一定程度影响了种子的语法完整性, 但是这也提高了生成种子的多样性. 本文需求目标是提高种子在模糊测试中的表现, 而存在一定非法性的种子更有可能探索到目标程序更多的执行路径甚至引发程序崩溃. 综上而言, 两种采样策略都是对种子语法合法性以及种子多样性的一种权衡, 从而更好的提高模糊测试的效果.

5 实验评估

我们进行了一系列的实验评估了我们系统对于提高种子文件在目标程序上模糊测试的代码覆盖率的提高程度.

5.1 实验环境

我们用于模糊测试的目标程序是 mupdf 软件 (1.4.0 版本). 我们提取了由爬虫在网上爬取的 37 628 个 PDF 文件作为原始种子语料库, 再提取其中所有长度小于 50 的 obj 对象, 对象的个数为 878 649 个. 其中 800 000 个 obj 对象用于作为训练集, 70 000 个 obj 对象用于开发集, 8649 个 obj 对象用于测试集. 这些 obj 对象经过格式处理后, 成为 obj 对象序列, 作为 obj 对象生成器中的 Transformer 网络的输入, 供网络学习和输出新的 obj 对象序列. Transformer 网络的训练和预测的实验环境为: Ubuntu16.04 的操作系统, 该系统下有 NVIDIA GTX1080 GPU、i7-7700 处理器和 16 GB 的 RAM. 网络的参数为: 模型的大小为 256 维, multi-head attention 隐藏层的层数为 8 层, 前馈神经网络为 1024 维, batch size 取 16, drop out 取 0.1, 序列最大长度为 60. Epoch=10 的模型训练时间为 9 小时.

在 Transformer 网络模型训练完毕之后, 我们使用训练好的模型进行预测. 我们将测试集中 8649 个 obj 对象序列的开头作为预测网络的输入, 输出经过网络预测的完整的 obj 对象序列. 再从测试集中的原始种子输入文件随机选取 10 个完整的 PDF 文件, 提取他们的文件结构. 最后将 obj 序列和文件结构在 PDF 组装器中组装成新的 PDF 种子文件, 作为最终模糊测试的种子输入.

5.2 代码覆盖率

我们测量种子语料库的代码覆盖率, 在本实验中, 我们将 mupdf 视为目标程序, 并比较了原始种子语料库和我们框架生成的种子语料库的代码覆盖率.

由于训练轮数是神经网络中的重要参数, 我们进行了 5 组对比实验, 分别取 epoch=10, epoch=20, epoch=30, epoch=40, epoch=50. 分别比较原始 PDF 文件和我们生成的新的 PDF 种子文件的代码覆盖率. 以 epoch=40 为例, 我们将原始种子语料库, Sample 采样策略+Transformer 网络下生成的新的种子文件, SampleFunc 采样策略+Transformer 网络下生成的新的种子文件, 进行了 24 小时的模糊测试, 最终得到的代码覆盖率的比较曲线如图 4 所示.

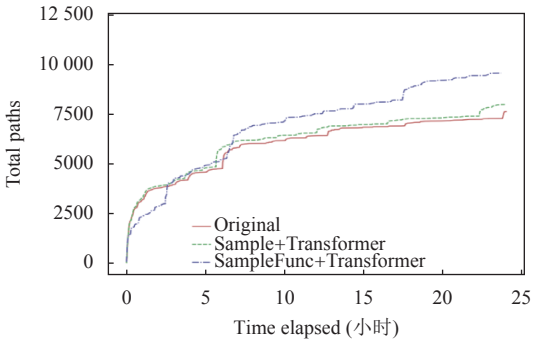


图 4 Epoch=40 下路径覆盖数比较

图 4 中, Original 表示原始种子语料库; Sample + Transformer 代表 Sample 采样策略下以及利用 Transformer 网络最后生成的新的 PDF 种子文件的代码覆盖率. SampleFunc + Transformer 代表 SampleFunc 采样策略下以及利用 Transformer 网络最后生成的新的 PDF 种子文件的代码覆盖率.

通过图 4, 我们可以发现无论是 Sample 采样策略下还是 SampleFunc 采样策略下, 由 Transformer 网络构成的 obj 生成器所产生的新的 obj 对象在组装成完整的 PDF 之后, 经过模糊测试得到的代码覆盖率相比原始种库都有了一定的提升, 特别是 SampleFunc 采样

策略下 Transformer 网络生成的 PDF 种子文件, 不仅能以更快的速度到达覆盖率增长的瓶颈, 并且也表现出了更高的覆盖率上限. 这表明, 此种方法的模糊测试效果最好.

训练轮数 epoch=10, epoch=20, epoch=30, epoch=40, epoch=50 下的本文框架生成的 PDF 种子文件在模糊测试上的表现情况如表 1 所示.

表 1 不同训练轮数下生成的 PDF 种子文件和原始种子文件的总路径数比较

策略	初始	Epoch				
		10	20	30	40	50
Sample	8506	8231	8840	8912	8966	8729
SampleFunc	8506	8177	9396	9941	11 974	8856

从表 1 中可以看出, epoch=10 时, 两种采样策略下的总路径数都比原始种子的代码总路径数要低, 原因可能是我们的网络训练的不充足. epoch=20 和 30 时, 总路径数都有了一定的提升. Epoch=40 时的效果最好, epoch=50 时, 虽然还是比原始种子的总路径数高, 但是较 epoch=40 时已经有了下降. 结合数据进行合理分析之后的我们的初步结论是, 由于训练轮数过多, 造成了一定的过拟合的现象, 导致模型的表征能力不增反降. 但总体而言, 实验证明, 在我们的框架下, 总路径数相较原始种子语料库都有了一定的提升.

5.3 采样策略效果对比

本文在 4.2 节中从理论上分析了采样策略的原因以及必要性, 但是这还需要实验结果的进一步证明. 我们将不使用任何采样策略生成的种子、采用 Sample 采样策略生成的种子、SampleFunction 采样策略生成的种子作为输入, 训练轮数 epoch=30, 对比 24 小时的模糊测试对比实验, 以覆盖路径数作为指标, 评估 3 种采样策略下种子的质量高低. 如图 5 所示, NoSample 即不使用任何采样策略, 它生成的种子覆盖路径数最少, 只有 3716 个, 而本文提出的两种策略下生成的种子覆盖路径数分别为 8912 个和 9914 个, 较之不采样方案的覆盖路径数多了很多.

我们以文本形式打开生成的 PDF 种子文件发现, 不采样策略下生成的种子中, obj 序列大量重复, 而其他两种采样策略下的 obj 序列, 没有任何两条是完全重复的. 结合这一点, 可以说明采样策略增加了 obj 序列的多样性, 而种子的多样性对于模糊测试而言十分重要, 因为重复的种子是完全冗余无用的. 这也证明了本文采样策略的有效性和必要性.

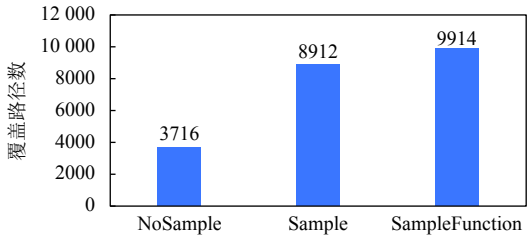


图 5 不同采样策略下种子覆盖路径数

5.4 种子大小

原始种子语料库中的种子大小从 0 KB 到 2228 KB 均匀分布, 而我们生成的种子大小有 39.05% 小于 3 KB, 最大的种子不超过 200 KB. 这比原始种子语料库中的种子小了很多. 因为我们只选取了长度小于 50 的 obj 对象序列作为训练数据, 所以我们得到的 obj 序列长度都不会太长.

5.5 其他格式性能分析

本文的优化框架是选择了 PDF 文件作为模糊测试的种子文件, 选择 PDF 文件的理由是因为 PDF 是复杂二进制文件格式的代表, 它其中包含了文字、图片甚至音频、视频等. 如果本文框架可以对 PDF 格式的文件有良好效果的话, 可以推断本文研究在别的简单格式文件下也具有一定的通用性.

我们进行了实验来探究这一点. 针对 PNG 格式, 训练 Transformer 模型来生成新的种子文件, 训练时间为 9 小时, 其他参数配置与 5.1 节中完全相同. 我们以 LoadPNG 作为目标程序, 将两种采样策略生成的种子与初始种子作为输入, 进行 24 小时模糊测试, 评估 3 类种子的覆盖路径数, 实验结果如图 6 所示, 可以看到本文研究提出的两种采样策略结合的 Transformer 模型生成的 PNG 种子文件, 覆盖的路径数比原始种子文件多, 这说明本文研究对于别的格式例如 PNG 格式, 也能起到不错的效果, 这也更说明了本文研究的有效性.

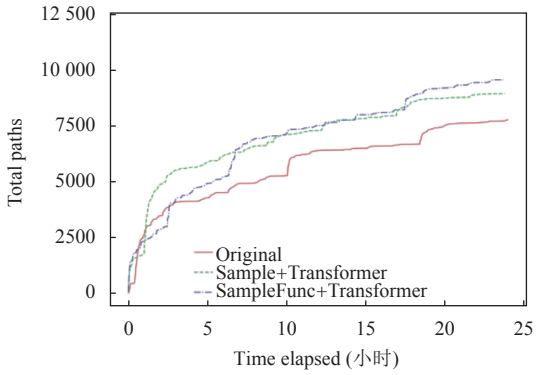


图 6 不同采样策略下的路径覆盖数比较

6 总结与展望

本文提出了一个利用机器学习来生成新的 PDF 种子文件的系统框架. 该框架学习并发掘 PDF 文件中 obj 对象的格式化文本信息, 并利用学到的知识智能地指导生成新 PDF 文件, 用于最后的模糊测试中. 实验结果表明我们生成的 PDF 文件, 不仅尺寸更小, 并且在更小的尺寸下, 在模糊测试中的代码覆盖率也有了较为明显的提升, 说明我们生成的种子在更小的尺寸下却有更高的表征能力.

但是, 我们的工作仍有需要改进的地方. 首先, 我们所使用的神经网络模型为 Transformer 模型, 然而还有很多擅长处理序列问题的神经网络模型如 RNN 模型以及它的多种变体. 我们可以将这些网络作为我们框架的训练网络, 作对照试验. 选取最优秀的网络模型作为 obj 对象的生成网络. 其次, 本文框架只是针对模糊测试种子质量进行了优化, 由之前的分析知道, 改变变异策略也可以提升模糊测试的代码覆盖率, 我们可以将本文框架与改变变异策略的方案结合起来, 进一步的提高对模糊测试的优化效果, 这可以作为我们未来的工作.

参考文献

1 McNally R, Yiu K, Grove D, *et al*. Fuzzing: The state of the art. Australia: Defence Science and Technology Organisation, 2012.

2 Godefroid P, Levin MY, Molnar D. Automated whitebox fuzz testing. Proceedings of NDSS ’2008 (Network and Distributed Systems Security). San Diego, CA, USA. 2008. 151–166.

3 Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery. Hoboken, NJ, USA: Addison-Wesley Professional, 2007.

4 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. IEEE Transactions on Software Engineering, 2019, 45(5): 489–506. [doi: 10.1109/TSE.2017.2785841]

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

6 King JC. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394. [doi: 10.1145/360248.360252]

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

8 Lv C, Ji S, Li Y, *et al*. SmartSeed: Smart seed generation for efficient fuzzing. arXiv preprint arXiv: 1807.02606, 2018.

9 Sutskever I, Martens J, Hinton G. Generating text with recurrent neural networks. Proceedings of the 28th International Conference on Machine Learning. Bellevue, WA, USA. 2011. 1017–1024.

10 Kurach K, Andrychowicz M, Sutskever I. Neural random-access machines. arXiv preprint arXiv: 1511.06392, 2015.

11 Reed S, de Freitas N. Neural programmer-interpreters. arXiv preprint arXiv: 1511.06279, 2015.

12 Parisotto E, Mohamed AR, Singh R, *et al*. Neuro-symbolic program synthesis. arXiv preprint arXiv: 1611.01855, 2016.

13 Bhatia S, Singh R. Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv: 1603.06129, 2016.

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

15 Pu YW, Narasimhan K, Solar-Lezama A, *et al*. sk_p: A neural program corrector for MOOCs. arXiv preprint arXiv: 1607.02902, 2016.

16 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 (ASE). Urbana, IL, USA. 2017. 50–59.

17 Nichols N, Raugas M, Jasper R, *et al*. Faster fuzzing: Reinitialization with deep neural models. arXiv preprint arXiv: 1711.02807, 2017.

18 Shi ZQ, Lin HB, Liu L, *et al*. FurcaNet: An end-to-end deep gated convolutional, long short-term memory, deep neural networks for single channel speech separation. arXiv preprint arXiv: 1902.00651, 2019.

19 Goodfellow IJ, Pouget-Abadie J, Mirza M, *et al*. Generative adversarial networks. Proceedings of the Advances in Neural Information Processing Systems. New York, NY, USA. 2014. 2672–2680.

20 Vaswani A, Shazeer N, Parmar N, *et al*. Attention Is all you need. arXiv preprint arXiv: 1706.03762, 2017.