软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种面向模糊测试的 GUI 程序空转状态实时检测方法*

张 兴[1], 冯 超[1], 雷 菁[1], 唐朝京[1]

[1](国防科技大学 电子科学学院,湖南 长沙 410072)

通讯作者: 张兴, E-mail: zxhree@hotmail.com

摘 要: 针对当前 Windows 下 GUI 软件模糊测试过程中由于进入空转状态时刻判断不准确导致的测试效率降低的问题,本文利用自然语言处理方法的在函数执行迹的基础上来解决空转状态识别问题.论文首先分析了传统程序分析方法在空转状态判断上遇到的困难,提出了基于 Bi-Gram 模型以及统计分析的空转状态识别方法.通过 Bi-Gram 算法将程序函数执行迹转换为概率特征序列; 利用空转状态在特征序列中的方差特征将空转状态特征序列从程序特征序列中分离,在此基础上进一步提取空转状态特征并实现空转状态实时检测算法.通过对典型源码与二进制软件程序的实验测试表明,本方法在效率和准确性上较传统方法上更优,能支撑对GUI程序模糊测试的需求.

关键词: 模糊测试;Bi-Gram 模型;GUI 程序测试;空转状态识别

中图法分类号: TP311

A Real Time Idle State Decection Method In Fuzzing

Zhang Xing[1], Feng Chao[1], Lei Jing[1], Tang ChaoJing[1]

[1](School of Electronic Science and Technology, National University Of Defense Technology , ChangSha 410073, China)

Abstract: GUI program's idle state usually cause the low efficiency of fuzzing test.It tries to solve idle state detecting problem based on function trace by nature language processing method.First it analyzes the difficulties that traditional program analysis method faces in idle state detection,and then proposes a idle state detecting method based on Bi-Gram module and statistical analysis.Bi-Gram algorithm transforms the function trace of the GUI program to probabilistic characteristics sequence.Then divides the idle state probabilistic characteristics sequence from prgram's probabilistic characteristics by variance characteristics of idle state probabilistic characteristics sequence and finally extract idle state features which applied to the real-time idle state detecting algorithm.Experiments of source code and binary program shows that new method is more efficient and accuracy than traditional method.

Key words: fuzzing test;Bi-Gram module; GUI program testing;idle state detection

二进制软件漏洞挖掘是网络空间安全领域研究的核心内容,而模糊测试则是软件漏洞挖掘中最广泛、最有效的方法,据统计当前 90%的未公开漏洞来自于模糊测试[1].

当前主流的二进制软件模糊测试方法主要针对系统配置、数据解析等类型的控制台程序,模糊测试引擎将生成的大量畸形测试数据通过文件、命令行或者网络接口输入给控制台程序,并监控程序的退出行为: 如果程序在处理过程中发生异常退出,则本次测试结束且发现漏洞;如果程序正常退出,则本次测试结束且未发现漏洞.

除控制台程序外,网络空间还存在以阅读器、办公软件等为代表的 GUI 程序.与控制台程序不同的是,GUI 程序在数据处理完毕后并不会退出,而是在用户图形上进行空转循环,等待用户的下一步操作指令.由于 GUI 程序在数据处理后不会退出运行,模糊测试引擎无法确认被测程序的数据处理过程是否结束,无法判断本次测试过程是否完成.

* 基金项目: 国家自然科学基金(61602502);国家重点研发计划(2016QY07X1500).
Foundation item: National Natural Science Foundation of China (61602502); National Key R&D Plan(2016QY07X1500).

空转循环状态,简称空转状态,是指 GUI 程序完成文件处理任务后会进入以消息循环为主的等待状态.如果测试引擎对程序进入空转状态时机判断不准确,就不能及时结束测试过程,降低模糊测试的准确率和效率.针对该问题,各类模糊测试引擎的主要采用解决方法有超时时限监测和 CPU 负载监测等方法.例如,AFL[2]、VUzzer[3]、Grinder[4]等当前最先进的模糊测试工具中采用超时时限检测方法.在该方法中,超时时限数值由用户根据经验手工设置. 当测试引擎发现被测程序在该数值后仍未异常退出时,则认为本次测试结束且未发现漏洞,强制结束被测程序的本次运行并开始下一次测试.虽然该方法在效率上存在较大缺陷,但是由于其实现简单并且误报率较低,在工业界得到广泛的使用.PEACH[5]工具则采用的 CPU 负载监测方法.该方法认为测试过程是否结束与 CPU 运行负载存在一定的关系,在 CPU 负载低于一定值且被测程序未异常退出时,测试引擎认为对测试数据的处理过程结束,会强制结束被测程序的本次运行并开始下一次测试.由于不同程序数据处理过程复杂度不同,且与运行的硬件性能关联紧密,基于超时时限监测和 CPU 负载监测的方法存在严重的问题:要么数据处理过程尚未完成而测试过程被强行结束,导致发生漏判;要么数据处理过程早已完成,而测试引擎未能及时发现,导致测试效率降低.现阶段来说,由于 CPU 负载监测方法实现较为复杂,且要求系统只能运行单一用例,浪费系统资源,效率较低,同时漏报误报率较高,该方法基本已经被工业界所抛弃.

通过对超时时限监测和 CPU 负载监测方法的研究,我们发现这些方法都是根据空转状态的外部表现进行判断,未抓住空转状态的内部本质特征,对空转状态判断不准确,导致模糊测试效率低.针对这个问题,本文深入分析 GUI 程序在空转运行状态,使用模糊匹配方法解决了空转状态识别问题.论文首次基于 Bi-Gram 模型建立了空转状态的程序函数执行迹模型,用概率特征值表示程序执行过程中消息循环的特征,并利用统计分析方法提取出程序进入空转状态的特征,最后提出了实时判定程序是否进入空转状态的快速检测算法,解决了模糊测试中对程序进入空转状态时机判断不准的问题.通过对真实案例的测试表明,本方法比现有的超时时限和 CPU 负载监测方法具有更高的准确性与效率.

论文在第 1 节介绍了 GUI 程序空转状态识别问题,第 2 节叙述了基于 Bi-Gram 模型以及统计分析方法的空转状态识别方法;第 3 节详细介绍了基于函数的 Bi-Gram 模型的建模过程,并给出了将函数执行迹转化为特征序列的方法;第 4 节介绍了利用统计分析的方法将特征序列中空转状态序列与非空转状态序列分离的方法;第 5 节介绍在空转与非空转序列的基础上进一步提取出程序进入空转状态的概率;第 6 节实现了空转状态实时检测工具,并通过一个验证实验与五个对比试验证明了空转状态实时识别方法的有效性与高效性.

1 GUI 程序空转识别问题

1.1 空转识别与软件模糊测试

图 1展示了空转状态对模糊测试效率的影响,由图可知,由于空转状态的存在,导致程序在完成测试用例处理后无法终止,浪费了计算和时间资源,降低了模糊测试效率.

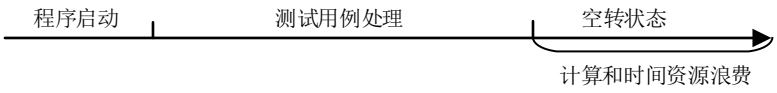


图 1 空转状态存在对模糊测试效率的影响

针对以上情况,业内普遍采用超时时限监测和 CPU 负载监测方法.超时时限检测是指,测试程序在监测到被测程序运行的时间超过预设的阈值时间后,强行终止被测程序,其中阈值时间由测试人员根据经验设置.这种方法会产生如图 2所示的问题,即针对简单测试用例时,被测程序在测试用例处理完后空转一段时间才会到达阈值时间,导致测试效率降低;而对复杂测试用例,可能出现被测程序还未处理完测试用例就到达时间阈值,导致被强行终止,发生漏报情况.

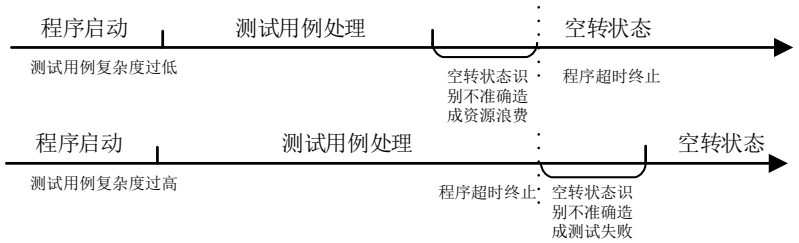


图 2 超时时限方法对模糊测试效率的影响

为了避免超时时限监测方法带来的模糊测试效率下降,诸如 Peach 模糊测试工具则采用了 CPU 负载监测的方法.该方法基于程序在测试用例处理过程中 CPU 占用率比较高,而空转时 CPU 占用率较低的原理.虽然该方法对部分程序有效,但是仍旧存在问题.首先,CPU 占用率并不能准确反应程序执行情况,在处理复杂度低的测试用例时,会出现CPU占用率过低而导致误报情况;其次,一些程序在空转状态时仍旧会有诸如定时的网络发送,后台写文件等行为,会出现在空转状态 CPU 占用率较高而导致漏报情况.最后,由于测试环境的特异性,部分CPU 占用率的波动并不是被测程序带来的,从而直接导致测试失败.

以上两种方法虽然在一定程度上解决了程序空转带来的模糊测试效率降低问题,但是由于没有从识别出程序空转的本质特征,只能依赖人工经验或者其他没有强因果关系的外部特征,导致针对模糊测试的效率仍旧不高.

1.2 GUI程序空转识别的复杂性

对于一般 Windows 下带 GUI 界面的软件来说,为了保证界面的流畅运行,都会采用多线程技术,将界面更新与数据处理分开处理.该流程的函数执行迹如图 3所示.

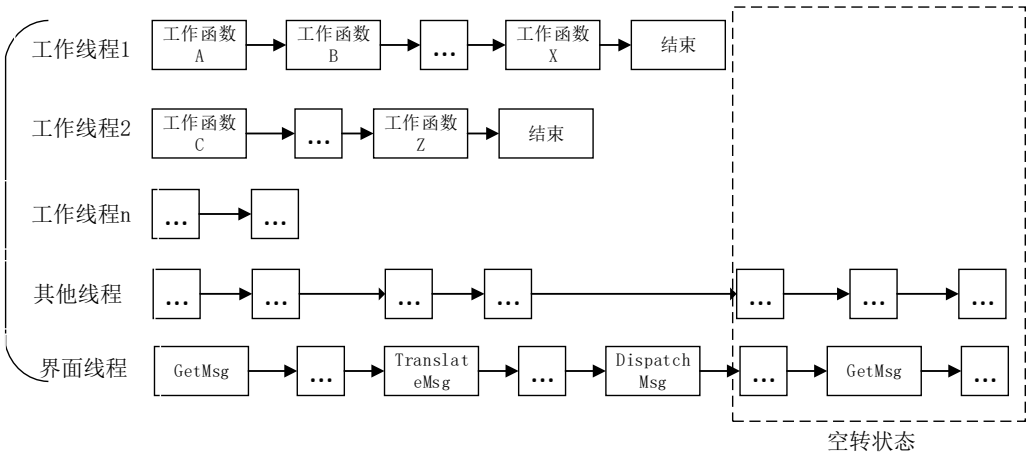


图 3 GUI 程序运行线程函数执行顺序图

界面线程在加载完成基本界面之后,基本上就处于 GetMessage→TranslateMessage→DispatchMessage 循环执行状态[6],这个状态被称为消息循环;其他线程则在整个程序运行过程中进行一些其他事物的处理,如定时器或者广告显示等,这些线程既不参与消息循环,也不会进行测试用例处理;工作线程通常负责测试用例的处理.在所有工作线程完成测试用例处理工作后,界面线程与其他线程仍旧处于函数循环执行的状态,该状态则被称为空转状态.

空转状态识别问题就是在被测 GUI 程序运行过程中准确地检测出程序进入空转状态时机.针对该问题可以采用两个手段: 一是单独分析每个线程,找到工作线程全部结束的时刻,并将其作为进入空转状态的时刻.使

用这种方法存在如下问题,首先根据测试用例的复杂程度不同,工作线程的线程数可能是不确定的;其次由于其他线程与界面线程共存,无法有效定位工作线程;最后部分程序会出现边处理边渲染的情况导致工作与空转状态交叉出现,会导致误判情况发生.

第二个手段是直接分析程序的函数执行迹.使用这种方法得到的函数执行迹流图会如图 4所示,函数 X 代表工作线程和其他线程的函数.

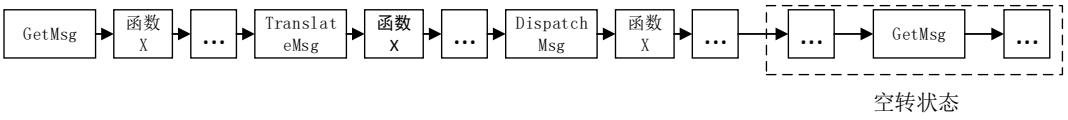


图 4 GUI 程序运行不区分线程函数执行顺序图

此外,在多线程程序中,线程的并发执行也将使相同程序每次执行的函数顺序具有一定的随机性,进一步导致空转状态的消息循环具有不确定性,使得每一个消息循环在整体看上去都是相似的,但是仔细对比发现个别函数在执行顺序有所不同,如图 5所示.

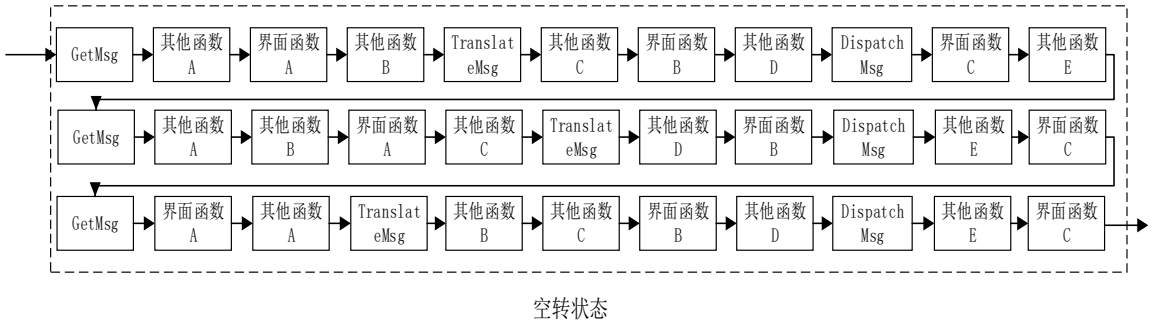


图 5 GUI 程序空转状态不区分线程流程图

更进一步,基于函数执行迹的空转状态识别的目的就是根据程序的函数执行迹,发现其中整体相似的消息空转循环,忽略异同的单个事件.识别效果可用两个准则判断：准确率和效率,如图 6所示.准确率是指识别方法在程序进入空转状态后识别出程序进入空转状态次数与总识别次数的比值.延误率是指识别出的进入空转状态的时机与程序实际进入空转状态时机的差值占程序实际进入空转状态时机的比值,而效率=1-延误率.

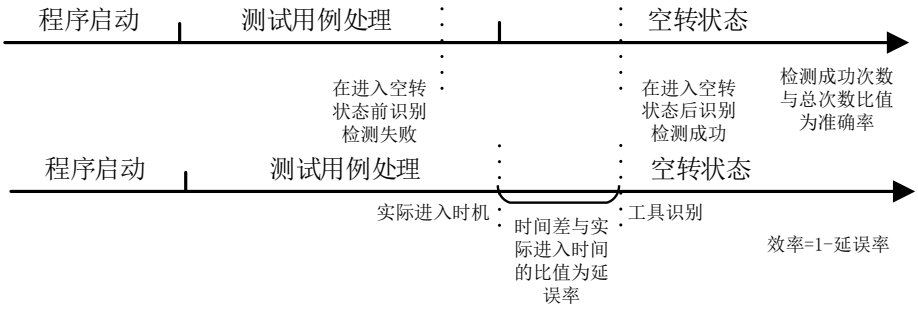


图 6 准确率和效率示意图

2 空转状态识别方法总体思路

针对空转状态识别面临的种种问题,本文提出了基于 Bi-Gram 模型[7]以及统计分析的空转状态识别方法.

总体方法的流程图如图 7所示,首先通过大量收集正常测试下目标程序函数执行迹,利用特征提取方法将这些函数执行迹转变为特征序列;然后通过统计分析方法将特征序列中空转特征序列与非空转特征序列分离,在此基础上提取空转状态特征;最后将空转状态特征作为参数传入空转状态实时检测算法中,实现空转状态实时检测.

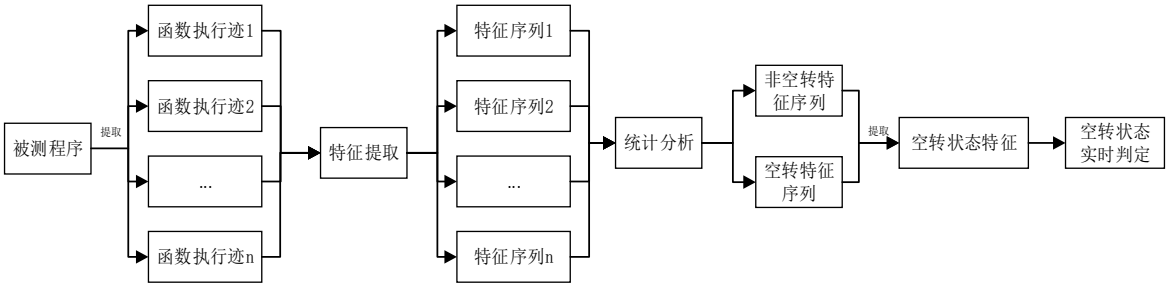


图 7 总体流程图

特征提取主要目的是将函数执行迹转化为用数值表示的特征序列.包括函数执行迹的提取、消息循环概率特征值的计算、语料库的收集以及空转状态的提取.训练阶段主要将程序的函数执行迹分割为以消息循环执行迹集合,在这些消息循环中提取元组并构建语料库.Bi-Gram 算法利用提取的语料库,将消息循环执行迹转变为概率特征值.最后,多个消息循环概率特征值按顺序排列,形成该程序函数执行迹的概率特征序列. 概率特征序列提取方法的工作流程如图 8所示.之后在程序概率特征序列基础上,根据空转状态概率特征的统计特性将程序概率特征序列中的工作序列与空转序列分开,并进一步提取出空转状态的特征.

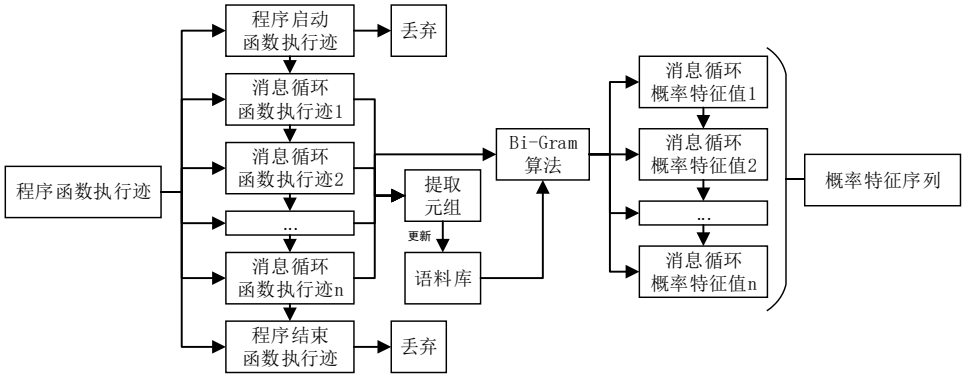


图 8 概率特征序列提取方法流程

实时检测主要目的是实时检测程序是否进入空转状态.实时检测算法利用空转状态特征,在程序运行时检测程序是否进入空转状态,实时检测算法工作原理如图 9所示.

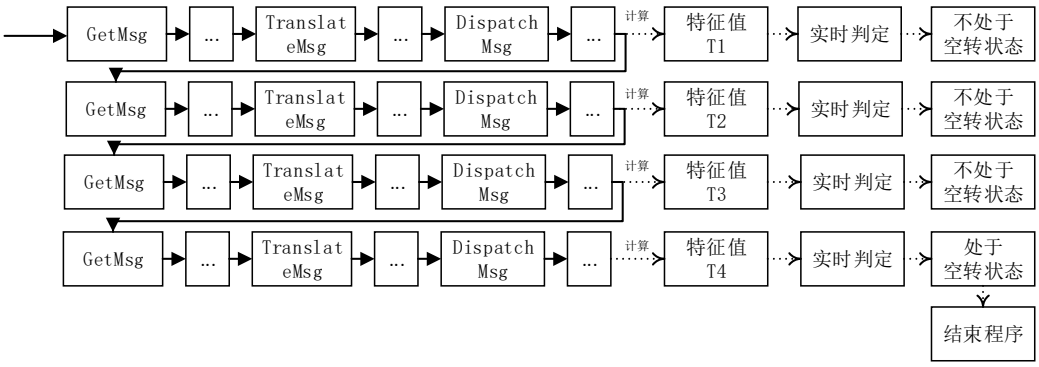


图 9 实时检测方法流程图

3 函数执行迹的 Bi-Gram 模型

由于多线程资源竞争,导致空转状态函数执行迹具有整体相似,个体相异的特点.为了解决该问题,可以将空转状态中的消息循环函数执行迹看成是函数状态间的转换,函数间执行顺序的不确定性可以用不同状态间转换的概率进行描述,而每个消息循环就可以用一个概率特征值来表示,从而将函数执行迹间的相似性用数值大小来衡量.转变的思路如图 10所示.因此,需要一个能够用数值方法描述状态序列相似度的模型来描述程序的函数执行迹.

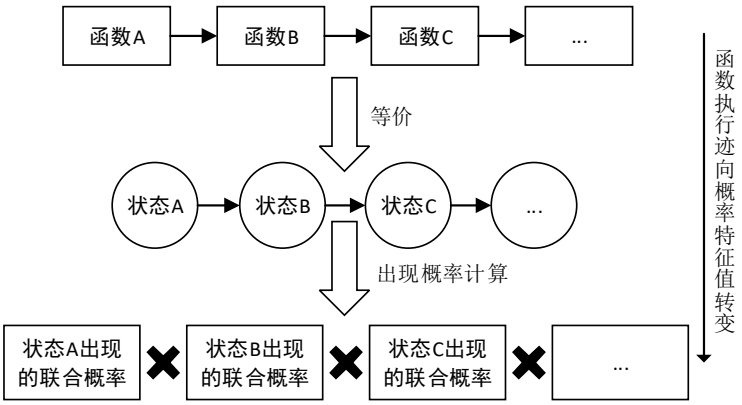


图 10 函数执行迹向概率特征值的转变

N-Gram 模型是一种基于统计的 N-1 阶 Markov 模型,通过描述前 N-1 个状态元组的出现对第 N 个状态出现的影响,计算整个状态集合出现的概率,从而描述多个状态序列间的相似度.该模型对状态序列的处理过程与问题解决思路基本一致.因此,本文使用函数执行迹的 N-Gram 模型实现执行迹序列到数值特征的转换.

在使用 N-Gram 模型建模前,首先要确定分隔符.对于一般 N-Gram 模型来说,其处理场景大部分为文字类,通常将空格或者标点符号作为分隔符,分隔符通常由待分析目标的特性决定.分隔符的选择通常需要与实际问题相结合[8],本文期望解决程序进入空转状态时识别不准确的问题,而空转状态与程序的消息循环密切相关,因此分隔符的选择需要与消息循环的典型特征相关.对于一般的 GUI 程序来说,消息循环由 GetMessage 类函数、DispatchMessage 类函数以及 TranslateMessage 类函数构成,而一个消息循环都是以 GetMessage 类函数判定为开始,DispatchMessage 或者 TranslateMessage 类函数在一个消息循环中调用的次数可能并不是固定的.因此,为了将消息循环有效分隔,选用 GetMessage 类函数作为基于函数执行迹的 Bi-Gram 模型的分隔符.

除了分隔符的确定,N-Gram 模型的 N 值的确定也是必要的.一般来说,模型阶数 N 刻画了待分析系统的结构,高阶模型能够更好地表述相应的系统模型[9],但是其空间复杂度和计算复杂度更高,严重影响计算效率,因此需要选取适当的模型阶数.对于程序的函数执行迹来说,模型阶数 N 主要的含义是第 N 个函数的出现只与前面已经执行的 N-1 个函数相关,也就是说,模型所描述的系统中出现 N 个状态组成的元组的次数是最高的.由于采用不区分线程的方式记录函数执行迹,导致某些函数的出现的先后顺序是随机的, N≥3 个固定函数组合出现次数不高,因此如果采用 N≥3 的模型阶数,将不能准确描述函数执行顺序的随机性.为了保证模型描述的准确性,选用 N=2 的 Bi-Gram 模型.

在正式介绍基于函数执行迹的 Bi-Gram 模型相关算法前,必须要了解语料库的构造.语料库就是存储待分析目标基本序列组合统计信息的数据库.与针对文本的语料库类似,基于函数执行迹的 Bi-Gram 模型语料库主要存储每种函数的频次以及最小状态转移元组的频次.在一个语料库中,如果某个状态组合出现的次数非常多,那么其概率特征值将会变得比较大,而出现次数较少的状态的概率特征值将会显得比较小,因此可以通过数值上的差异来区分这两个状态.要想实现较好的空转状态区分效果,就需要构造以空转状态为主的语料库.本文采用的构造方法是,让被测程序加载测试用例,并使该程序长时间处于空转状态,提取其函数执行迹.并将该过程重复多次,提取多条函数执行迹构造以空转状态为主的语料库.

在进一步介绍本文所涉及的理论前,首先确定一些定义.本文将函数用 f 表示,消息循环的函数执行迹用 G 表示,有 $G=f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_n$.程序的函数执行迹用 T 表示,有 $T=G_1\rightarrow G_2\rightarrow\ldots\rightarrow G_x$,其中下标 n 表示程序运行的第 n 个函数,其中 n>=1,且 n 为自然数.

对于一段函数执行迹 $f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l$,采用 Bi-Gram 模型进行建模,就是将执行迹中函数的执行近似为 1 阶 Markov 模型.也就是说,对于拥有 l 个函数的函数执行迹 $f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l$,只有前 1 个函数对下一个函数出现的概率有影响,用概率表示是:

$$P(f_l \mid f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_{l-1})\approx P(f_l \mid f_{l-1}) \tag{1}$$

根据乘法定理和 Bi-Gram 模型的分析原则[10],函数执行迹 $f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l$ 的出现概率可表示为组成该执行迹的函数的概率的乘积:

$$P(f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l)=\prod_{i=1}^{l}P(f_i \mid f_1\rightarrow\ldots\rightarrow f_{i-1})\approx\prod_{i=1}^{l}P(f_i \mid f_{i-1}) \tag{2}$$

由于 $P(f_i \mid f_{i-1})$是可以根据执行迹在训练语料库出现的频率来估计得到:

$$P(f_i \mid f_{i-1})=\frac{C(f_{i-1}\rightarrow f_i)}{C(f_{i-1})} \tag{3}$$

函数 $C(x)$表示变量 x 出现的次数.这里 $C(f_{i-1}\rightarrow f_i)$表示 Bi-Gram 模型元组 $f_{i-1}\rightarrow f_i$ 在函数执行迹中出现的次数,$C(f_{i-1})$表示函数 f_{i-1} 在函数执行迹中出现的次数.

因此,对于消息循环 $G=f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l$,其概率特征为:

$$P(G)=P(f_1\rightarrow f_2\rightarrow\ldots\rightarrow f_l)=\prod_{i=1}^{l}\frac{C(f_{i-1}\rightarrow f_i)}{C(f_{i-1})} \tag{4}$$

由于在实际情况下,一个程序的消息循环执行迹包含的函数个数 l 通常比较大,导致求得的 $P(G)$值过小,甚至无法表示,因此本文采用取对数的方法表示消息循环的函数执行迹的概率特征:

$$N(G)=\lg(P(G))=\sum_{i=1}^{l}\lg(\frac{C(f_{i-1}\rightarrow f_i)}{C(f_{i-1})}) \tag{5}$$

由此,可得对于由 n 个消息循环执行迹组成的函数执行迹 $T=G_1\rightarrow G_2\rightarrow\ldots\rightarrow G_n$,其概率特征序列可表示为 $N(T)=N(G_1)\rightarrow N(G_2)\rightarrow\ldots\rightarrow N(G_n)$.

4 GUI 程序空转特征提取

本节利用统计分析方法将 $N(T)$中的空转与非空转状态分离,即确定在该次程序执行过程中,程序经历了多少次消息循环后进入了空转状态.

在以空转状态为主体的语料库的基础上,程序在进入空转状态前处理事物种类较多, $N(G)$值波动较大,而进入空转状态后,$N(G)$值则相对平稳.其可能的情况如图 11所示.由于数据的波动性可以用方差来表示,则程序进入空转状态时刻之前 $N(G)$序列的方差与之后 $N(G)$序列的方差的差距是最大的,基于此提出了从函数执行迹中分离出空转状态执行迹的方法.

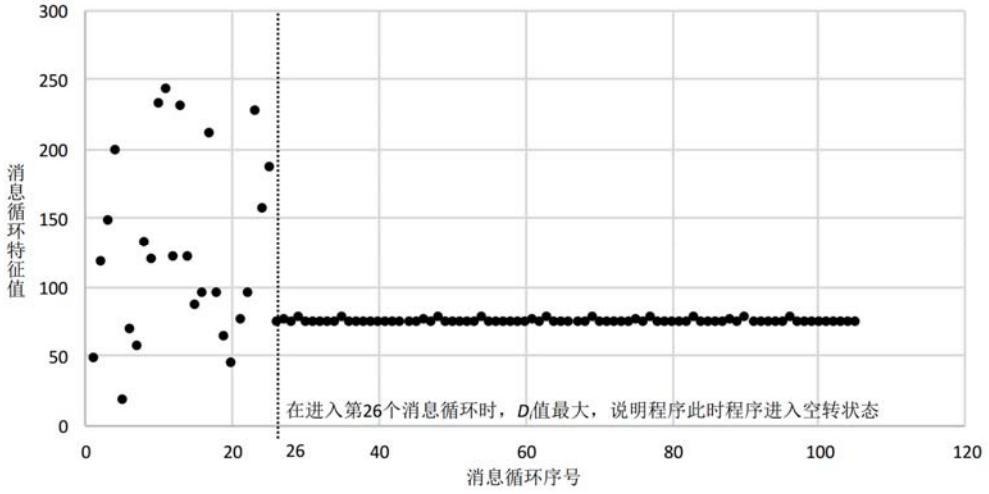


图11 函数执行迹N(G)图

假设现有函数执行迹概率特征为 $N(G_1)\rightarrow N(G_2)\rightarrow\ldots\rightarrow N(G_N)$,分别计算前 i 个特征值的方差 $D_f(i)$,以及后 N-i 个特征值的方差 $D_b(i)$,其中 $1\le i\le n$-1.此时计算前 i 个特征值的偏差与后 N-i 个特征值偏差的差值 $D_i=D_f(i)-D_b(i)$,代表以 i 为分界线前后特征值的偏差情况.找到 D_i 中值最大的那个 D_{imax},表示以 $imax$ 为界限,前后的偏差是最大的,说明 $imax$ 前的特征偏差很大而 $imax$ 后的特征值偏差很小,符合程序进入空转状态的情况,因此确定程序在该次运行时是在第 $imax$ 次消息循环中进入空转状态的.由图 11可知,经过计算,其在 i=26 时 D_i=4276.337 值最大,因此,对于该特征序列, $imax$=26.经过与表格数据比对,符合真实情况.

对于非空转状态特征序列 $N(T_N)=N(G_1)\rightarrow N(G_2)\rightarrow\ldots\rightarrow N(G_{imax})$以及空转状态特征序列 $N(T_I)=N(G_{imax})\rightarrow N(G_{imax+1})\rightarrow\ldots\rightarrow N(G_l)$,在此基础上需要进一步提取出空转状态概率特征值.由于空转状态的判定需要实时进行,因此判定过程计算复杂度一定比较低.本文定义三个变量 *IdleNGAver*、*Range* 以及 *IdleCountThreshold* 来描述空转状态特征.*IdleNGAver* 参数表示空转状态 $N(T_I)$集合的平均值;*Range* 参数表示 *IdleNGAver* 在空转状态 $N(T_I)$集合中的最大偏差;*IdleCountThreshold* 参数表示在非空转状态 $N(T_N)$集合中,出现的 $N(G)$值在 *IdleNGAver*±*Range* 范围内的次数.这三个变量计算方式如下:

$$IdleNGAver = \frac{\sum_{i=i\max}^{l} N(G_i)}{l - i\max}$$

$$Range = Max(\{|N(G_i) - IdleNGAver|, i\max \le i \le l\}) \quad (6)$$

$$IdleCountThreshold = C(\{N(G_i) \mid 0 < i < i\max, |N(G_i) - IdleNGAver| < Range\})$$

空转状态 $N(G)$值由于待测程序运行的特异性和随机性导致其不是特定的散列值,而是在 *IdleNGAver* 上下波动的范围值,为了降低漏报率,将进入空转状态后的在 *IdleNGAver* 上下波动范围的最大值作为程序进入空转状态时的范围值,即 *Range* 值. 被测程序在运行时若 $N(G)$值与 *IdleNGAver* 的差值在 *Range* 范围内,即可说此时程序已进入空转状态.然而由于 *Range* 值可能会将非空转状态的 $N(G)$值包含进去,从而会产生误报,为了避免此类情况,算法加入了次数阈值变量 *IdleCountThreshold* 值,用于统计程序在非空转状态 $N(G)$值在 *Range* 范围内的个数.被测程序运行时若 $N(G)$值在 *Range* 范围内次数超过 *IdleCountThreshold*,则说明此时程序已经进入空转状态.本文通过设置 *Range* 值来降低漏报率,通过加入 *IdleCountThreshold* 来降低误报率.在实际测试中,通常会多次测试被测程序,选取其中最大的 *IdleCountThreshold* 以及 *Range* 来进一步保证低漏报率与误报率.

对于图 11 中的特征值图的情况,如图 12 所示,图中所示程序 *IdleNGAver* 为 76,*Range* 值为 5,*IdleCountThreshold* 值为 2.

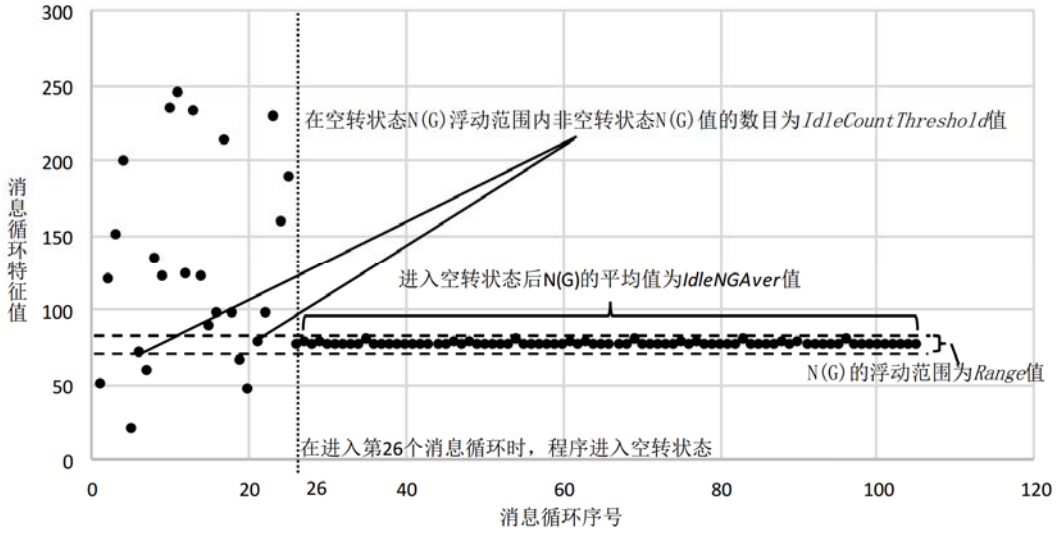


图 12 利用第 5 节示例提取空转状态特征

5 空转状态的实时判定

空转状态实时检测就是在程序运行时,根据程序函数执行过程,实时生成 $N(G)$值,并根据 $N(G)$值是否进入空转状态进行实时判定.由于 $N(G)$值的计算仅仅涉及到整数加及内存查找运算,对被测程序运行效率几乎没有影响; $N(G)$值计算完成后就需要判定程序是否进入空转状态,为了尽量降低判定过程对被测程序运行效率的影响,判断过程的时间复杂度一定要低.

由于 Bi-Gram 模型以及语料库的选择,程序的空转状态与非空转状态在 $N(G)$值上必然存在数值上的差异,因此,程序进入空转状态时,消息循环的 $N(G)$值必然在空转状态概率特征值范围内.基于此原理,本文提出的空转状态实时判定算法如下:

```
BOOL isInIdleState(double NG)  // 函数参数N(G)表示当前求得的消息循环的N(G)值
```

```
{
    if(abs(NG-IdleNGAver)<Range){
        idleCount++;
        if(idleCount >= IdleCountThreshold){
            return True;    }   } //判定进入空转状态
    return False;}
```

对于消息循环执行迹来说,其函数个数 l 并不是个定值,这也就导致了可能会出现两个完全不同状态的消息循环的 $N(G)$值相同的情况;同时,由于程序在运行时可能会出现先加载界面再加载测试用例的情况,导致本应属于空转状态的 $N(G)$出现在非空转状态 $N(G)$集合里,因此,在检测时,需要一个累计变量 *idleCount*,检测出现 $N(G)$的次数超过一定阈值后,判定程序进入空转状态.

在实时判定算法的基础上,进一步可以得到实时检测算法.算法在程序运行时利用语料库以及三个特征值根据程序实时函数执行迹判断程序是否进入空转状态. 实时检测算法如图 13所示.

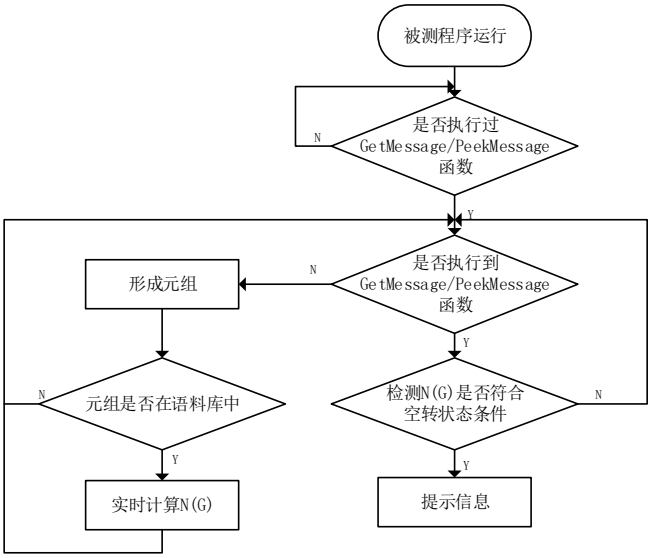


图 13 实时检测算法流程

由图可知, 实时检测算法在程序运行时实时构造元组数据,并采用实时检测算法判断当前 $N(G)$是否满足空转状态条件.当算法检测到形成的元组不在语料库中时,说明该元组是不属于空转状态的,因此直接跳过元组生成等工作,等待下一个消息循环的进入,以此提高程序的执行效率.

6 实现与实验

本章实现了 Windows 下 GUI 程序空转状态实时检测原型工具.通过一系列实验验证了本文提出空转状态识别方法的准确性以及效率,首先本文选用了修改过源码的带 GUI 程序 TestApp,用来测试算法的准确性;然后选择了五个真实存在的应用程序,通过模拟模糊测试的超时时限过程与使用算法进行比较,验证方法的效率.

6.1 工具实现

工具分为三部分,分别为函数执行迹记录部分、特征提取部分以及实时检测部分,整体关系如图 14所示.

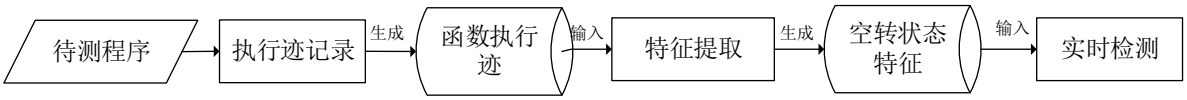


图 14 工具整体关系

对于 Windows 下带 GUI 界面的程序来说,程序的运行过程不仅仅是跟程序有关,同时也跟相关的动态链接库有关.实际上,消息循环的主体函数大都是系统自带动态链接库的 API 函数,因此,对程序的函数执行迹的记录,不仅仅需要记录程序自身内部的函数,对其所涉及的动态链接库的函数也要进行记录.由于本文分析的主要目标是程序本身的行为,对动态链接库相关的内部行为并不关心,因此对于动态链接库,函数执行迹的记录主要是针对动态链接库中的导出函数,而忽略其内部函数;对于程序本身来说,函数执行迹的记录主要是其内部自定义的函数.通过记录动态链接库的导出函数以及程序内部自定义函数执行轨迹,来表示程序消息循环的特征.本文所需要记录的函数如下表 1所示:

函数类型	是否需要记录
程序自身内部函数	√
动态链接库导出函数	√
动态链接库内部函数	×

表 1 执行迹记录部分所需记录的函数

本文采用了 PIN[11]作为执行迹记录的平台,因此,动态链接库的导出函数可以直接通过符号加载器监控.然而,PIN 对程序自身定义的函数却没有办法进行监控,因此,本文首先使用 IDA PRO 对被测程序进行分析,生成自定义函数数据库.

模块在被测程序运行前首先对程序自定义函数进行地址转换.由于在 Windows 7 或更高的系统中,加入了地址随机化(ASLR),因此,程序自身的加载基址会随着程序每一次执行而改变,因此,本文在程序加载进入内存中时,记录其加载基址,将自定义函数数据库内的地址进行转化,从而执行迹记录模块可以在程序运行时正确地找到程序自定义函数的入口.

特征提取模块主要有两个作用,一是生成及更新基于 Bi-Gram 模型的语料库,二是生成及更新空转状态判断特征.语料库提取功能首先读取函数执行迹数据库,然后以 GetMessage/PeekMessage 为分隔符将函数执行迹分割为“句子”,最后,按照 Bi-Gram 模型提取出 $C(f_{i-1}\rightarrow f_i)$以及 $C(f_{i-1})$,并计算出 $lg(P(f_i|f_{i-1}))$,将其保存在语料库中.

语料库的更新过程,语料库内主要保存三种数据,$C(f_{i-1}\rightarrow f_i)$、$C(f_{i-1})$以及 $lg(P(f_i|f_{i-1}))$,假设原预料数据库内有 $C(f_{i-1}\rightarrow f_i)_{before}$、$C(f_{i-1})_{before}$ 以及 $lg(P(f_i|f_{i-1})_{before})$,现加入一个新执行迹的数据,新执行迹提取出 $C(f_{i-1}\rightarrow f_i)_{now}$、$C(f_{i-1})_{now}$ 以及 $lg(P(f_i|f_{i-1})_{now})$,则更新过后的语料数据库内有 $C(f_{i-1}\rightarrow f_i)_{new}=C(f_{i-1}\rightarrow f_i)_{now}+C(f_{i-1}\rightarrow f_i)_{before}$、$C(f_{i-1})_{new}=C(f_{i-1})_{now}+C(f_{i-1})_{before}$ 以及 $lg(P(f_i|f_{i-1})_{new})=lg(C(f_{i-1}\rightarrow f_i)_{new}/C(f_{i-1})_{new})$.通过这种方法完成语料库的更新.

针对多个 $N(T)$,本文选用对 *IdleNGAver* 取平均,*Range* 和 *IdleCountThreshold* 取最大值的方式实现空转状态特征值的更新.由于不同的 *IdleNGAver* 在同样的语料库下算得的差距并不会太大,因此采用取平均的方法;而 *Range* 表示属于空转状态特征的情况,为了尽可能的将满足空转特征的概率特征加入到判别阈值中,选用最大的 *Range*;同样由于采用保守策略,尽可能降低误报率,选取最大的*IdleCountThreshold*.

实时检测模块在程序运行时利用语料库以及三个特征值实时判断程序是否进入空转状态.实时检测模块首先加载语料库以及自定义函数库,然后进行地址转换,最后开始验证工作.检测算法流程如图 13所示.

6.2 实验与结果分析

本节介绍一个验证实验和一个对比实验.验证实验采用自行开发的的带 GUI 程序 TestApp,TestAPP 实现了对文件的加载,并将文件的内容通过一定的运算,将运算结果显示到窗口上,该程序包含一个消息循环,文件处

理过程采用单独的线程进行计算,同时,TestAPP 还在程序运行时统计执行过消息循环的个数,并在程序完成文件加载运算以及显示后停止计数,并将该计数作为结果输出.然后,使用空转状态识别算法对该程序进行空转状态识别,同样记录程序被识别进入空转状态时所经历的消息循环的个数.通过对比两个消息循环的个数来判断算法的准确度.

对比实验选用五款商用软件,分别是 FoxitReader、WPS 表格、WmDownLoader、VUPlayer 以及 IKEView.对这五款软件采用空转状态识别算法,在识别到程序进入到空转状态时结束被测软件进程;同时,采用 PIN 加载这五款软件,分别统计这五款软件从启动到界面稳定所需要的时间,并将该时间设置为超时时限.在相同的时间内,模拟模糊测试流程,检测两种方法运行程序的次数,以此来估算空转状态识别算法为模糊测试带来的效率,证明了本文所提的方法相比传统时间阈值方法更加有效.

6.3 实验结果

6.3.1 验证实验结果

通过不同次数的学习对比,检测算法判断出程序进入空转状态时所经历的消息循环次数,并与程序本身的进入空转状态所经历的消息循环次数进行比较.通过使用不同大小的文件进行测试,检测工具能否有效地区分待测程序的工作状态与空转状态;使用的执行迹条数表示语料库构造时用到的函数执行迹个数;TestAPP 记录结果表示程序完成测试用例处理后进入空转状态时所经历的消息循环的个数,该功能在 TestAPP 源码上实现,其记录结果表示真实情况;工具实验结果表示工具记录的程序完成测试用例处理后进入空转状态时所经历的消息循环的个数,每次测试完成后,会同时产出 TestAPP 记录结果与工具实验结果,保证测试环境相同.实验结果如图 15所示.

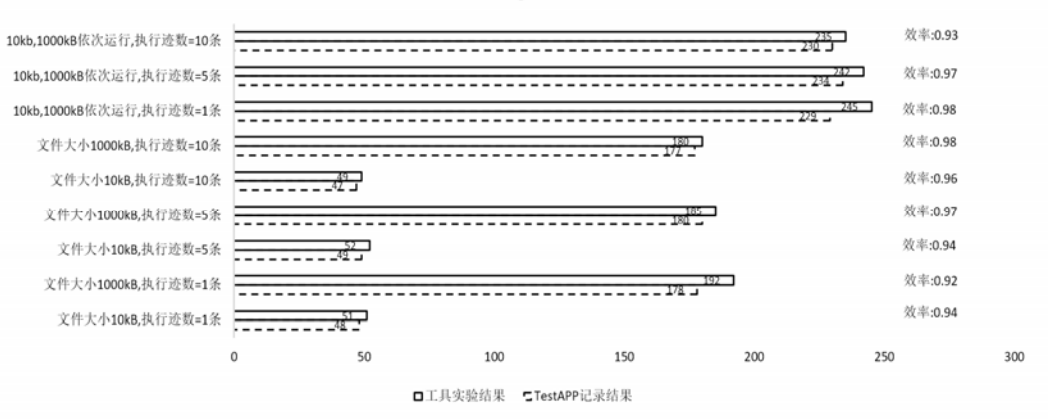


图 15 验证实验实验结果

由结果图 15可知,使用的执行迹条数越多,生成的语料库更能反映空转状态特征,算法测出的被测程序进入空转状态所经历的消息循环次数与实际所经历的消息循环次数越来越接近.同时,测试结果显示识别出空转状态的时刻是进入空转状态后,表示工具的准确率高.不同大小文件的依次运行,表明虽然测试用例的复杂度不同,工具仍能准确识别出程序进入空转状态的时机.随着构造语料库的函数执行迹的增多,工具的效率有一定程度的提高,说明构造丰富的语料库可以有效提升工具的效率.

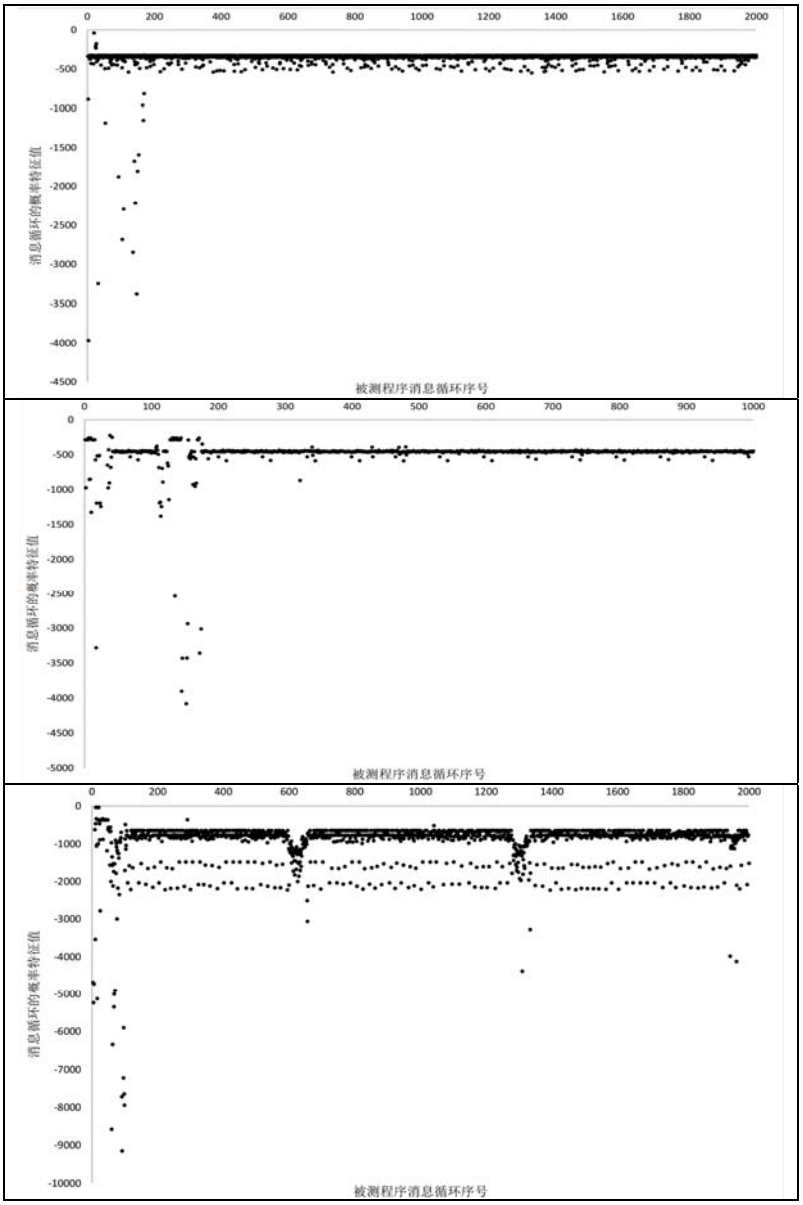
6.3.2 对比实验结果

分别针对 IKEView、WmDownLoader、VUPlayer、FoxitReader、 WPS 表格、FlashFXP 以及 Dupsctc 七种软件进行对比实验.IKEView 在读取文件前需要有 MessageBox 按钮点击确认过程;WmDownLoader 必须通过 GUI 交互完成文件输入;VUPlayer 可以使用命令行完成文件输入;FoxitReader 和 WPS 表格则是当前具有代表性的大中型商业软件;FlashFXP 以及 Dupsctc 则是从网路上获取数据并处理的.这七款软件中,IKEView、

WmDownLoader、VUPlayer、FoxitReader 以及 FlashFXP 选用 GetMessage 函数作为分隔符;由于 WPS 表格以及 Dupsctc 采用 QT 编写,其消息循环依靠 PeekMessage 实现,选用 PeekMessage 作为分隔符.在进行测试前,每个程序加载测试用例,并等待 20 分钟记录函数执行迹,往复运行 10 次,得到语料库.

为了检测在测试过程中是否会出现误报问题,即待测程序进入空转状态前进程被杀死.本文所选用的程序都有相应的 POC,通过运行 POC 检测崩溃界面是否出现来判断工具是否存在误报.

首先展示经过预处理后,每个程序从启动加载测试用例到关闭的消息循环的 $N(G)$值的散列图.



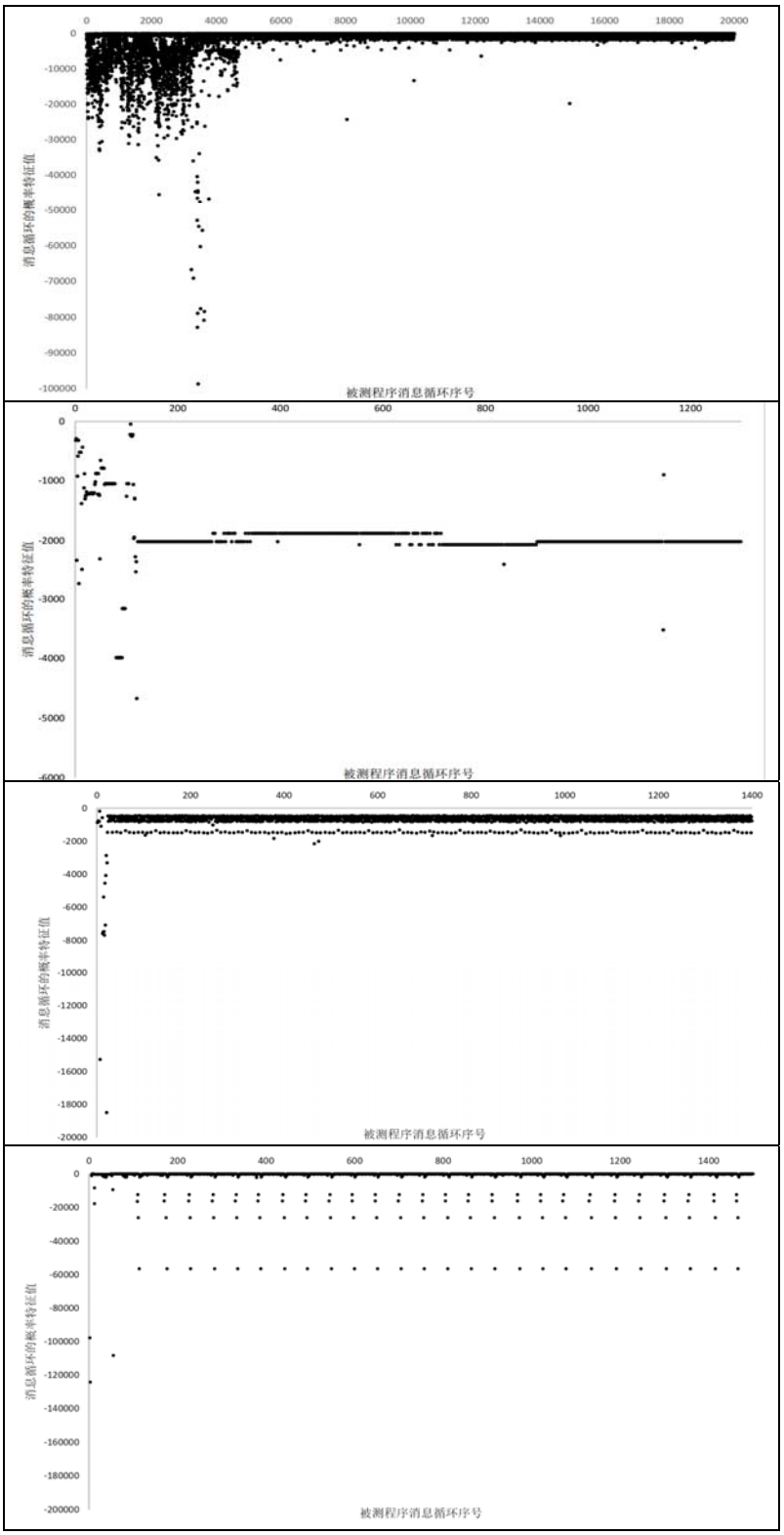


图 16程序启动到关闭的消息循环概率特征序列,其中上到下依次为 VUPlayer、WmDownLoader 、

FoixtReader、WPS 表格、IKEView、FlashFXP 以及 Dupsctc.

由图可知,这些程序的 $N(T)$图都具有如下特征：开始分布发散,而后聚敛.符合第 4 节的推断.

经过实际检测,被测程序在加载 POC 时,都正常弹出崩溃界面或者触发异常调试器,结果表明工具具有高准确率.通过与超时时限方式进行对比,即执行相同多次数的程序,比较所消耗的时间.为了控制相应的变量,被测组采用 PinTool 实时检测工具进行加载,对比组采用 PIN 进行加载,通过观察的方式得到超时时限值.为了尽可能还原工业界在超时时限方法的处理方式,即保证程序在完成测试用例用例处理后结束程序,本文超时时限的阈值时间的选择是随机测试 20 个大小不同的测试用例,选择处理时间最长的那个作为阈值时间.每个程序分别加载 300 个不同大小的测试用例,记录所需时间(单位：秒),记录结果如图 17所示

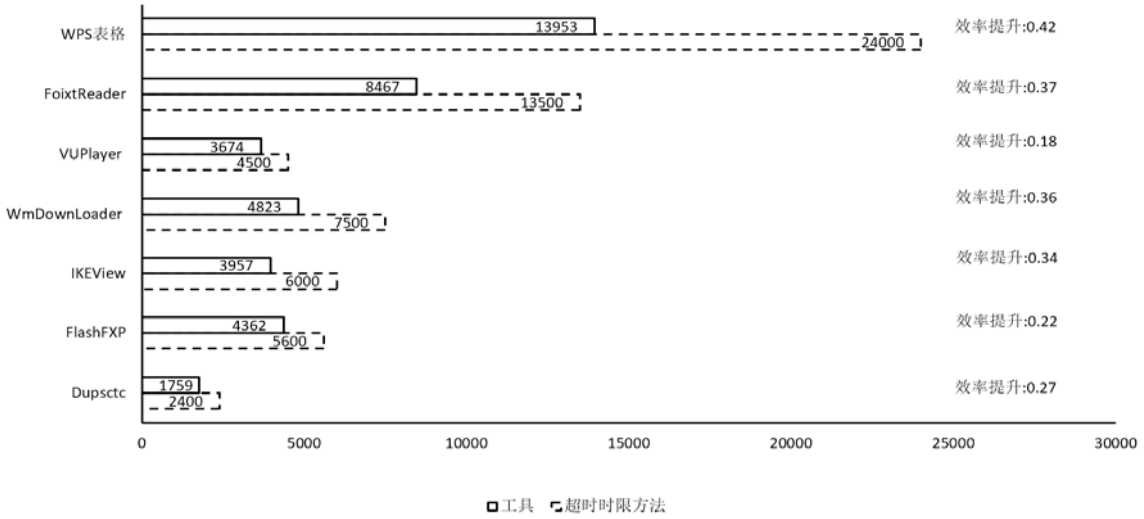


图 17 对比实验实验结果(单位:秒)

由图中数据可知,由于测试用例文件大小不一,因此使用超时时限方法时为了避免文件加载完成前结束程序,因此需要选取耗时最长的时间作为超时时限参数时间,因此会出现测试用例已经加载完毕,但是仍旧需要等待超时时限时间的情况,从而降低了测试效率.而本文提供的工具则是通过对程序本身状态进行判断,测试用例大,则执行时间长,测试用例小,则执行时间短.因此,在面对规模较大的程序时,能够准确对程序进入空转状态的时机做出判断,从而提升测试的效率.

7 总 结

本文提出了基于函数执行迹的 Bi-Gram 模型来判断程序进入空转状态的时机,从实验结果看,原型工具在识别准确与效率上都比现阶段流行的超时时限方法优秀.接下来,根据原型工具,可以进一步应用到 Windows 下的 DBGENG 下,与现有的模糊测试工具中的后端异常检测相结合,进一步实用化,提升模糊测试效率.

同时,也可以将模型应用到除空转状态的其他的程序状态识别,例如,测试用例文件加载或者测试用例文件处理等行为,这样可以减少人工程序分析的难度,来提取程序关键功能区域.

References:

[1] http://www.freebuf.com/articles/neopoints/111712.html

[2] http://lcamtuf.coredump.cx/afl/ American Fuzzy Lop

[3] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[J]. 2017.

[4] https://github.com/stephenfewer/grinder An Automate the Fuzzing of Web Browsers System.

[5] Schachter K. Peach fuzz[J]. Long Island Business News, 2005.

[6] Russinovich M, Solomon D A. Windows internals: including Windows server 2008 and Windows Vista[M]. Microsoft press, 2009.

[7] Sharma A, Lyons J, Dehzangi A, et al. A feature extraction technique using bi-gram probabilities of position specific scoring matrix for protein fold recognition[J]. Journal of theoretical biology, 2013, 320: 41-46.

[8] Wu Yingliang, Wei Gang, Li Haizhou,et al. A WORD SEGMENTATION ALGORITHM FOR CHINESE LANGUAGE BASED ON N-GRAM MODELS AND MACHINE LEARNIN, Journal of Electronics and Information Technology, 2001, 23(11):1148-1153.

[9] MAO Wei, XU Wei-ran, GUO Jun,A Chinese Text Classifier Based on n-gram Language Model and Cha in Augmented Naive Bayesian Classifier,JOURNAL OF CHINESE INFORMATION PROCESSING Vol. 20 No. 3

[10] Cavnar W B, Trenkle J M. N-Gram-Based Text Categorization[C]// In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval. Las Vegas, US. 1994:161--175.

[11] Pin: Intel’s Dynamic Binary Instrumentation Engine, Pin Tutorial[J]. Intel Corporation, 2010.

附中文参考文献:

[8] 吴应良, 韦岗, 李海洲. 一种基于 N-gram 模型和机器学习的汉语分词算法[J]. 电子与信息学报, 2001, 23(11): 1148-1153.

[9] 毛伟, 徐蔚然, 郭军. 基于 n-gram 语言模型和链状朴素贝叶斯分类器的中文文本分类系统[J]. 中文信息学报, 2006, 20(3):29-35.