一种针对格式文件的符号执行优化方法

汪孙律[1,2],杨秋松[1],李明树[1]
(1. 中国科学院软件研究所基础软件国家工程研究中心,北京 100190;
2. 中国科学院大学计算机科学与技术学院,北京 101408)

摘 要: 为了解决符号执行中路径爆炸、新路径发现率低等问题,提出了针对文件格式数据块约束的符号执行分析方法(FFCBSE,File Format Constraint Based Symbolic Execution)优化框架. 文件格式信息的缺失会影响符号执行的效率以及测试用例生成,该方法通过分析程序代码自动分析程序读取的格式文件数据块之间的依赖关系并建立相关约束,随后使用这些约束引导符号执行更关注于核心功能代码区域. 在 KLEE 中实现了上述优化框架,并对 Tcpdump、Readelf、Elfdump、File、Zlib 等 7 个常用文件处理程序做了检测. 和 KLEE 以及 DASE 相比,FFCBSE 发现了 13 个之前未知的缺陷,在指令覆盖率和分支覆盖率有 10% ~ 225% 不同程度的提升.

关键词: 符号执行; 文件格式; 路径爆炸; 缺陷查找
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2020)12-2417-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2020.12.018

A Symbolic Execution Optimization Method Based on File Format Constraint

WANG Sun-lü[1,2], YANG Qiu-song[1], LI Ming-shu[1]
(1. *NFS, Institute of Software, Chinese Academy of Sciences, Beijing* 100190, *China*;
2. *School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing* 101408, *China*)

Abstract: To solve problems like path explosion, low rate of new path's finding in the software testing, a new vulnerability discovering architecture based on file format constraint (FFCBSE) was proposed. FFCBSE analyzed program source code to extract file structure constraints automatically. FFCBSE then used these structure constraints to guide symbolic execution to focus on core functions. This architecture was implemented in KLEE, and it was evaluated on seven file processing applications, such as Tcpdump, Readelf, File, Zlib. Compare with KLEE and DASE, FFCBSE detects thirteen previously unknown bugs. In addition, FFCBSE increases instruction line coverage/branch coverage by 10% ~ 225%.

Key words: symbolic execution; file format constraint; path explosion; bug finding

1 引言

符号执行是一种重要的形式化方法和代码分析技术,采用抽象符号代替程序变量,根据程序语义遍历程序的执行空间,从而收集执行路径上所有的约束条件. 该技术在软件测试和程序验证[1]中发挥着积极的作用,近年来还被应用于程序漏洞检测[2]、自动程序修复[3]等领域中.

符号执行基本过程如下:符号执行引擎首先将输入的具体值(concrete)替换为符号值(symbolic),并对符号值添加适当的约束. 当执行到程序的条件分支,并且该分支条件依赖于符号值时,引擎会计算该分支两个方向的可达性,并且沿着可达的分支添加新的符号状态到约束集中. 随着程序条件分支的增长,路径的规模会以指数级速度同步增长,最终导致路径爆炸问题. 其中一类解决方法是通过目的导向优化方法,引导执行引擎搜索未覆盖的指令,比如 Edmund Wong 等人提出了基于文档辅助的建模方法 DASE[4]. DASE 方法以可执行和可链接类型的文件(ELF, Executable and Linkable Format)为目标,通过自然语言提取规则从帮助文档中提取对文件结构体的描述,比如格式文件中起始字节需要等于某些固定值. 另外,如果处理对象是格式

收稿日期:2019-09-04;修回日期:2020-11-04;责任编辑:郭游
基金项目:中国科学院战略性先导科技专项(No. XDA-Y01-01)

文件,读取操作的处理还涉及到符号执行的环境建模问题,比如 KLEE[5] 将文件系统符号化,即每个状态都会包含多个由用户指定大小和数量的符号化文件,当通过系统调用读取某个文件时返回一个确定大小的符号值. 然而以上方法面临以下三个问题:(1)约束的提取依赖于帮助文档的语言描述方式和内容完整性,无法有效的适用于没有帮助文档的格式文件;(2)难以从文档中提取文件各部分数据的依赖关系生成的约束;(3)类似 KLEE 对文件系统整体符号化的方式过于粗粒度,容易带来路径爆炸的问题.

针对以上问题,本文提出了一种针对文件格式数据块约束的符号执行分析方法(FFCBSE, File Format Constraint Based Symbolic Execution),根据程序对文件句柄的处理流程,利用静态分析方法跟踪所有的读取文件的位置,建立不同位置的文件数据依赖关系树,随后自顶向下分层符号化读取的文件数据,即保留原始的文件格式上的依赖关系,又能提升特定功能模块的覆盖率,加速分析过程. 另外,本文还解决了部分格式文件解析类程序的环境建模问题. 先前研究中使用的文件系统建模方式过于粗粒度或者不具备通用性,而 FFCBSE 方法通过一种分析算法能够自动提取格式文件结构约束,做到文件的细粒度符号化,即不依赖于描述文档也不需要用户指定任何参数.

2 背景介绍及示例

传统的符号执行算法中,对文件输入的处理与普通的变量一致,比如经典的符号执行工具 KLEE 中,用户需要指定符号文件的个数和长度,在处理过程中有读取文件的行为则返回符号值代替实际读取数据. 标准的 elf 文件格式分别由 ELF 头(ELF header)、程序头表(program header table)、节(section)和节头表(section header table)4 部分,节数据又由文本(text)、只读数据(rodata)、数据(data)等内容组成. 文件结构中的 3 个依赖关系如下:(1)程序头表和节头表依赖于 ELF 文件头;(2)节数据依赖于节头表;(3)程序数据依赖于程序头表.

典型的节头数据处理流程如图 1 所示. 假设节头数据处理流程中存在缺陷,那么在达到该位置之前,搜索引擎需要越过图中文件格式处理和节头格式处理两个节点. 当前已有的符号执行工具无法有效的处理以上情况,是因为搜索过程中会遇到以下三个问题:(1)无法有效的将文件的数据依赖关系转换为相关的约束. 如图 1 中读取节头数据并对比长度,而参与对比的长度数据又来源入口处的 16 字节. 如果能够建立各域的依赖关系,有效的符号化数据将提升符号执行的效率;(2)无法准确限定文件必要的长度. 当

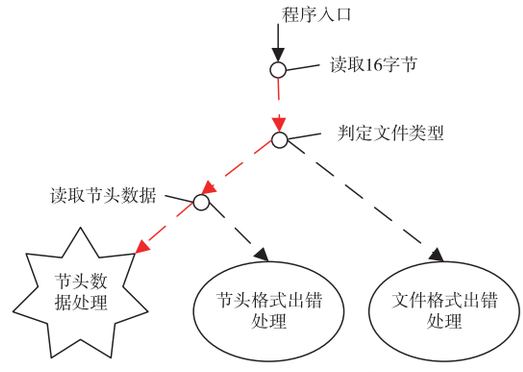

图1 节头数据处理流程图

前主流的符号执行工具对于内存的符号化方式即指定起始地址和长度,而文件类似的通过指定文件长度进行符号化,指定的长度过短则最终的生成的实例化结果可能不符合文件规范;(3)处理多个文件相关的依赖关系. 如果目标程序会一次读取多个文件,并建立各自的依赖关系,全部展开并自由组合进行符号化那么效率不高. 如果能够对各种组合进行简化,针对多个文件读取过程中的符号化进行优化,可以有效提高执行效率.

解决以上问题的关键在于对文件各数据域的依赖关系进行提取,并转化为符号执行工具可识别的数据结构. 为此本文设计了一套静态分析方法用于分析文件处理过程的执行路径,对可能的标识位置进行标记,同时建立读取数据块操作之间的依赖关系,随后通过一种迭代的算法对依赖关系进行简化,建立一组“完全依赖”的关系树,随后自顶向下进行分层次的符号化,从而最大程度的利用文件本身的结构特点,减少处理无效数据的资源损耗,同时设计了一种算法高效处理多个文件处理程序中的符号化流程. 本文贡献在于:

(1)提出了一种的新方法 FFCBSE 用于提升执行引擎的搜索效率. FFCBSE 提取输入文件的起始字节以及文件各部分的依赖关系生成相应的约束,在符号化过程中利用文件格式的约束信息减少无效路径的搜索,尽可能的覆盖文件处理相关联的核心功能代码.

(2)解决了部分格式文件解析类程序的环境建模问题. 先前研究中使用的文件系统建模方式过于粗粒度或者不具备通用性,而 FFCBSE 方法通过一种分析算法能够自动提取格式文件结构约束,做到文件的细粒度符号化,即不依赖于描述文档也不需要用户指定任何参数.

(3)基于开源工具 KLEE 实现了对 FFCBSE 的支持,可兼容其它 KLEE 的优化插件以及复用搜索策略.

3　基本定义和格式文件约束提取算法

3.1　基本定义

定义 1(目标文件操作标签)　一个测试程序可能会多次读取文件的不同位置,这些读取操作构成一张文件操作标签表 $\mathrm{FOPT}=\{\{\mathrm{FILE}_0,r0_0,r0_1\cdots r0_{pos}\},\{\mathrm{FILE}_1,r1_0,r1_1\cdots r1_{pos}\}\cdots\{\mathrm{FILE}_n,rn_0,rn_1\cdots rn_{pos}\}\}$,因程序可能会读取多个文件,所以该标签表是一个二维表,其中每行的第一参数表示读取的目标文件,ri_j表示对第 i 个文件的位置 j 的操作,同时也是一个普通的过程节点,描述见定义 3. 为方便描述,后文使用 FOPT[f]表示文件 f 的读操作集合.

定义 2(程序控制流图)　使用四元组〈N,E,entry,exit〉表示程序控制流图,其中 N 表示该执行过程的节点的集合,E 为是每个节点之间边的集合,entry 表示入口节点,exit 表示退出节点. 每个节点表示为顺序执行语句的基本块,其中 N 中节点和 entry 节点的最后一个语句为跳转语句. E 中每条边表示节点 n_i 和节点 n_j 属于 N 或者 entry/exit 节点,并且从节点 n_i 到节点 n_j 存在控制流转移的有序节点对〈n_i,n_j〉,满足 n_i 是 n_j 的直接前驱,n_j 是 n_i 的直接后继,每个节点可以有多个直接前驱节点和直接后继节点,entry 是一个没有前驱的节点,exit是一个没有后继的节点.

定义 3(文件操作节点)　使用三元组〈Start,Len,Data〉表示文件操作控制流图中的每一步操作,也就是操作节点,其中 Start 表示文件读取的起始位置,Len 表示读取的长度,Data 表示读取的数据集合. 为方便标识,后文简写为〈S,L,D〉. R'表示文件操作节点的集合,其中 R'是 N 的子集.

定义 4(文件操作控制流图)　文件操作控制流图 FOPT-CFG 可以由程序控制流程图 CFG 遍历生成,并用四元组〈R',E',entry,exit〉表示,其中 R'表示过程中文件操作节点的集合,每个操作节点表示对文件的某部分数据块做操作. E'是边的集合,entry 表示入口节点,exit 表示退出节点. 对于集合 R'中两个节点 r_i 和 r_{i+1},〈〈r_i,r_{i+1}〉〉表示两个文件操作之间存在执行路径,对于该节点关系必须满足以下条件:

$$\langle\langle r_i,r_{i+1}\rangle\rangle\in\text{FOPT-CFG}\equiv\langle r_i,n_{i+1}\rangle\wedge(\wedge_{i+1<k<j-1}\langle n_k,n_{k+1}\rangle)\wedge\langle n_j,r_{i+1}\rangle\wedge(r_i=n_i)\wedge(r_{i+1}=n_{j+1})$$

其中某个文件 f 对应操作控制流图,以 FOPT-CFG(f)表示.

定义 5(文件操作依赖运算)　本文使用运算符 $\prec$ 表示文件操作之间存在直接依赖关系. 对于集合中 R'中的两个节点 $r_i,r_j,r_i\prec r_j$的定义为对于文件操作 $r_j\langle S_j,L_j,D_j\rangle$和 $r_i\langle S_i,L_i,D_i\rangle$满足:

$$r_i\prec r_j\equiv(S_j\in D_i\mid L_j\in D_i)\wedge(r_i\in\mathrm{FOPT}(f))\wedge(r_j\in\mathrm{FOPT}(f))$$

即如果两个文件操作之间存在直接依赖关系,则需要这两个操作都属于同一个文件的操作控制流图,并且属性 S、L 与属性 D 之间有依赖关系,文件 f 由依赖关系构成的依赖图称之为 $\delta(f)$.

定义 6(文件操作完全依赖运算)本文使用运算符 $\lesssim$ 表示相邻操作之间存在完全依赖关系,在定义 5 的基础上,对于依赖图上任意相邻的两个文件操作 r_i 和 r_j,$r_i\lesssim r_j$定义为:

$$r_i\lesssim r_j\equiv(r_i\prec r_j)\wedge\nexists k\,(\wedge_{i<k<j}(r_i\prec r_k)\wedge(r_k\prec r_j))$$

通过完全依赖定义对文件操作依赖图中的可能存在的间接依赖操作做进一步的简化,从而减少符号执行过程的复杂度,在 $\delta(f)$ 基础上的完全依赖图称之为 $\Delta(f)$. 该过程反复迭代的,直到无法进一步简化.

定义 7(关联性文件操作)　如果一个程序需要处理两个文件,这两个文件的操作序列可能会有关联性,即某个文件的读取操作会依赖于另一个文件的读取结果,比如某个读取操作的起始地址来自于其中一个文件,而偏移量来自于另外一个文件. 如果对于文件 f_i 和 f_j,两个处理流程有关联性,σ 运算会返回所有关联操作的列表,其定义为:

$$(\mathrm{List}=\sigma(f_i,f_j))\equiv(r\in\Delta(f_i))\wedge(r\Delta(f_j))$$

3.2　约束提取算法

算法 1 为文件操作控制流程图生成算法,其中 3 ~ 5 行通过遍历控制流图中的所有路径中节点,并筛选出所有的文件操作节点. 随后在 6 ~ 9 行建立节点之间的依赖关系,并将该节点加入到文件操作控制流图中. 最后在第 14 行建立当前文件控制流图和文件标识之间的

算法 1　文件操作控制流图生成算法

输入:程序控制流图 CFG;目标文件操作标签 FOPT;目标文件标识 f.
输出:文件操作控制流图 FOPT-CFG.

```
1. FOPT-CPG-NODES←{EntryNode,ExitNode}
2. Parent←EntryNode
3. foreach Path in CFG
4.    foreach Node in Path
5.       if Node in FOPT or Node is ExitNode
6.          Parent. succ←Node
7.          Parent←Node
8.          if Node not in FOPT-CFG-NODES
9.    FOPT-CFG-NODES←Node ∪ FOPT-CFG-NODES
10.         end if
11.      end if
12.   end for
13. end for
14. FOPT-CFG←CfgGen(FOPT-CFG-NODES, f)
```

映射关系. 文件操作依赖图 δ 和完全文件操作依赖图 Δ 生成的过程类似,在此不再展开.

算法 2 为文件类型标识约束集生成算法,我们首先对中间语言行为进行抽象标识. 目标程序的中间语言的节点可以抽象为:

〈Modifier〉Dest = OP〈Modifier〉SRC1〈Modifier〉SRC2

其中 Modifier 指示目的操作数和源操作数的类型,OP 为对应的操作码. 对于一个字符串比较操作可由以下方式表示:

〈bool〉result = strcmp〈string〉filestr〈const〉Ident.

其中操作码为比较操作 cmp,第一个源操作数为字符串变量,Ident 为常量字符串用于对比操作,比较返回的结果为布尔类型.

算法 2 文件类型标识约束集生成算法

```
输入:程序控制流图 CFG;文件操作控制流图 FOPT-CFG;目标文件标识 f.
输出:文件类型标识约束,FileFormatConstraintSets 简称 FFCS.
1. for eachReadNode in FOPT-CFG
2.   if ReadNode is not ExitNode or EntryNode
3.     if ReadNode in FOPT-CFG-NODES
4.       if ReadNode. Start eq ZERO
5.         foreach Node in CFG
6.           if ((Node. op eq strcmp) and (Node. SRC1 eq Read-
               Node. Data))
7.     File Format-Nodes-Set←Node ∪ File Format-Nodes
8.           end if
9.         end for
10.      end if
11.    end if
12.  end if
13. end for
14. FFCS←ConstraintGen (FileFormat-Nodes-Set, f)
```

文件类型标识约束集生成算法遍历文件控制流图中的每个节点,如果该节点读取的是文件开头的数据(第 4 行),则从该节点开始遍历后续的所有的操作节点(5 ~ 9 行),如果该节点的操作为字符串比较操作,同时该操作的操作目标为文件数据(第 6 行),则将该节点加入到约束节点集合中. 最后在第 14 行将所有的节点生成约束加入到约束集中.

以下描述算法 3,完全依赖操作约束集生成算法的过程,过程中为提高符号化效率,以一组节点为粒度进行符号化,而不是以单个节点. 其中 7 ~ 11 行采用了广度优先的搜索方式,记录每层的节点,随后在第 14 行生成对应的约束集合. 24 ~ 26 行根据每组约束集开始符号执行流程.

如果存在关联性操作,即某个节点和两个文件相关联,其中当前遍历的文件操作完全依赖图对应的文件标识为主依赖,而相关联的文件称之为从依赖. 为了减少复杂度,从依赖只需要考虑相关联节点以及相关联节点的父节点生成的符号约束,并在主依赖图遍历到对应层级时加入到约束集合中.

相关联性操作的处理如算法 3 的 17 ~ 21 行所示所示,在遍历每层节点时都会检查是否有相关联节点存在,如果存在则生成对应的约束并加入到约束集合.

算法 3 完全依赖操作约束集生成算法

```
输入:文件操作完全依赖图 Δ;目标文件标识 f;关联文件操作集 σ;关联文件标识 r.
输出:约束集序列,ConstraintSets Sequence 简称 CSS.
1. Quetmp←Δ(f). Entry;Visited,CSS,count←ϕ
3.   START:
4.     Que←Quetmp; Quetmp←ϕ
6.     if Que not eq ϕ
7.       foreach Node in Que
8.         foreach ChildNode in Node. succ
9.           if ChildNode not in Visited
10.            Quetmp←Quetmp ∪ ChildNode;
11.            Visited←Visited ∪ ChildNode
12.            endif;
13.    endfor; endfor;
14.    FF CS←ConstraintGeneration(Quetmp, f)
15.    CSS←CSS ∪ FFCS; DEQEUE(Que,Node)
16.    foreach Node in σ(r)
17.      foreachChildNode in Node. succ
18.      if ChildNode in Quetmp
19.        Related-Node-Set←Related-Node-Set ∪ Node
20.      endif;endfor;endfor;
21.      FFCS←ConstraintGen(Related-Node-Set,r)
22.      JUMP START
23.    else
24.      foreach ffcs in CSS
25.        Start Symbolic Execution with FFCS
26.    endfor;
27. endif;
```

以上整个流程如图 2 所示,其中每张图的右半部分为文件 f 的完全依赖图,而左半部分为文件 r 的完全依赖图. 如图中框内标识的节点所示,文件 f 的约束集过程类似于自上而下的广度遍历. 如果遇到存在相关联性操作,即将文件 r 的节点也加入到约束集中,如图 2(a)中间框标识的部分. 在执行过程中,根据符号化节点的深度,低于该深度为实际执行,而高于该深度的为符号执行. 随着符号化过程的迭代,符号化的时机越来越晚,直到所有的文件读节点遍历完毕,整个过程如图 2(b) ~ (d)所示.

约束集算法的时间复杂度和文件格式中数据依赖的层数成正比关系,而空间复杂度并未有明显的增加. 根据

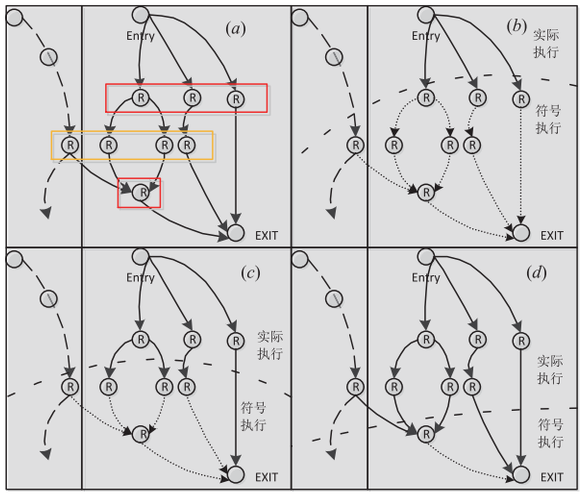

图2　迭代符号执行的过程

统计的数据,格式文件的依赖关系一般不超过 5 层,因此相比于传统建模的方法,本方法并不会增加过多的开销.

4　实验与分析

所有实验都是在处理器为 Intel Xeon X5675 CPU (24cores,3.07GHz)、内存为 94GB 的服务上进行,操作系统是 64 位的 Ubuntu 14.04 LTS.

FFCBSE 包括静态分析和动态分析两个模块,原型系统在 LLVM 上实现了静态分析环节,包括文件读操作判定、文件操作控制流图、完全依赖图、关联文件操作标识等工作,为动态分析模块提供辅助信息,动态分析模块在 KLEE 基础上实现,包括符号约束的插桩和后续的符号执行流程.

本文从开源工具中选择了 Tcpdump、Binutils (readelf)、elfToolChain (elfdump)、File、Zlib、Flac、ImageInfo 这 7 种不同的文件处理功能软件,基本信息如表 1 所示,描述了各个目标软件的版本、功能以及对应的静态指令数量. 实验针对 FFCBSE 的指令覆盖率、分支覆盖率、缺陷发现能力等方面进行了测试,对比的对象为业内流行的符号执行工具 KLEE 以及 DASE 方法,其中 DASE 只支持 ELF 格式文档的解析,所以对比对象为 Readelf 和 Elfdump 两款软件.

表 1　测试对象

名称	版本	简介	代码行
Tcpdump	4.9	网络数据包分析	47782
Readelf	2.14	elf 格式文件分析	25891
Elfdump	0.7.1	elf 格式文件分析	22534
File	2.3	识别文件类型	12597
Zlib	1.2.9	数据压缩函数库	7835
Flac	1.3.1	一种音乐压缩格式	46114
Imageinf	1.1.2	读取图片信息	56573

4.1　指令覆盖率和分支覆盖率

指令覆盖率和分支覆盖率是用于性能对比的两个指标数据. 指令覆盖率是度量被分析代码中每个可执行语句是否被执行到了,分支覆盖率则表示代码中的分支是否被执行. 本文使用 gcov 作为统计语句覆盖率的工具. KLEE 的主要参数设置如下:最大执行时间为 7200s 即 2h,最大求解时间为 80s,搜索策略为随机,与 DASE 方法的环境一致.

表 2 和表 3 描述了指令覆盖率和分支覆盖率的在 KLEE、DASE 和 FFCBSE 对比的情况. "*K*" 表示 KLEE, "*D*" 表示 DASE, "*F*" 表示 FFCBSE; *F*/*K* 表示相比于 KLEE 的提升比率,*F*/*D* 表示相比 DASE 的提升比率. 通过对比可以发现,相比于 KLEE,FFCBSE 在指令覆盖率和分支覆盖率有 10% ~225% 的提升. 和 DASE 相比,除 elfdump 的分支覆盖率基本持平,其它都有明显的提升.

表 2　KLEE/DASE/FFBSE 指令覆盖率对比图

名称	*K*%	*D*%	*F*%	Δ. PP(*F*/*K*)	Δ. PP(*F*/*D*)
Tcpdump	9.8	不支持	14.6	49.6	\
Readelf	6.9	15.2	22.4	225.4	47.5
Elfdump	16.1	22.1	26.9	66.9	21.6
File	13.8	不支持	16.3	17.5	\
Zlib	26.7	不支持	30.7	14.8	\
Flac	12.5	不支持	17.3	38.6	\
Imageinf	10.4	不支持	14.2	37.3	\
Elfdump	9.8	不支持	14.6	49.6	\

表 3　KLEE/DASE/FFBSE 分支覆盖率对比图

名称	*K*%	*D*%	*F*%	Δ. PP(*F*/*K*)	Δ. PP(*F*/*D*)
Tcpdump	8.7	不支持	12.0	39.0	\
Readelf	6.9	16.6	19.2	178.7	15.8
Elfdump	20.4	30.7	30.2	47.0	-1.3
File	11.1	不支持	14.2	28.3	\
Zlib	19.5	不支持	22.4	14.4	\
Flac	9.3	不支持	14.4	54.3	\
Imageinf	7.9	不支持	11.8	48.9	\
Elfdump	8.7	不支持	12.0	39.0	\

图 3 和图 4 分别给出了 tcpdump 在测试过程中指令覆盖率和分支覆盖率的实时变化情况,可以发现执行效率上 FCBSE 比 KLEE 的优化效果明显.

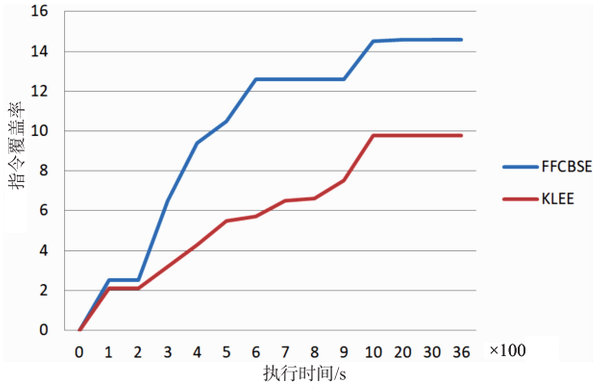

图3 指令覆盖率的执行速度

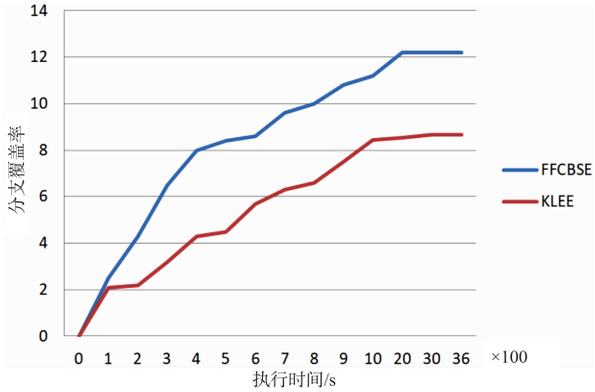

图4 分支覆盖率的执行速度

4.2 缺陷发现能力

在使用静态方法将文件格式的约束加入到符号执行之后,相比于 KLEE,FFCBSE 方法发现更多的程序缺陷. 在对 7 个偏重于文件处理的程序测试中,KLEE 能够发现 4 个程序缺陷,而 FFCBSE 可以发现 17 个缺陷. 另外根据 DASE 给出的实验结果,在对 Readelf 检测中,FFCBSE 能够多发现 2 个缺陷. 表 4 给出了对比结果.

表 4 缺陷发现能力对测试结果

程序	KLEE	DASE	FFCBSE	Δ(*F*/*K*)	Δ(*F*/*D*)
Tcpdump	2	不支持	9	+7	\
Readelf	0	1	3	+3	+2
elfdump	1	1	1	0	0
File	0	不支持	1	+1	\
Zlib	0	不支持	1	+1	\
Flac	1	不支持	2	+1	\
Imageinf	0	不支持	0	0	\

如图 5 所示,本文以 readelf 发现的 BUG 为例说明 FFCBSE 的缺陷发现能力.

该处缺陷为越界错误,该段代码存在于 process_dynamic_segment 函数中,如上图所示. 其中 17 行从文件中读取 edyn 数据,随后使用 edyn 中的 d_tag 作为索引读取 dynamic_info,通过对 edyn 数据符号化,就会发现该处的越界风险.

在执行 process_dynamic_segment 函数之前需要依次执行 get_file_header、process_file_header、process_program_headers 等多个功能函数,每个函数都会读取 ELF 格式文件关联的数据并做相应的处理. 比如 2 ~ 5 行必须满足其中一种标识才能通过格式检测. 同样的通过该方式,还可以跳过其他类型的文件格式约束检查,从而将搜索空间集中在核心功能区.

```
1.  process_file_header (file){
2.    if (e_ident [EI_MAG0] ! = ELFMAG0
3.      || e_ident [EI_MAG1] ! = ELFMAG1
4.      || e_ident [EI_MAG2] ! = ELFMAG2
5.      || e_ident [EI_MAG3] ! = ELFMAG3) {
6.        Error (_("Not an ELF file - ……"));
7.        return 0;
8.      }
9.  process_section_headers (file);
10. process_program_headers (file);
11. process_dynamic_segment (file) {
12.   get_64bit_dynamic_segment (file);
13.   dynamic_info[edyn. d_tag] = edyn. d_val;
14. }
15. get_64bit_dynamic_segment (file) {
16.   ……
17.   edyn = get_data(file);
18. }
```

图 5 缺陷代码

5 相关工作

目标导向优化是近年来符号优化的重要方向,它不是指特定的某种方法,更类似于一种思想[6~8]. 这种优化手段并不以覆盖全局为目标,而只对程序感兴趣的片段或者功能进行验证,比如针对程序的补丁、特定的应用场景、特定的应用目标等多个方向. 这类优化大多通过静态分析增加约束条件,引导符号执行工具集中在某些目标程序路径上. 比如文献[9]中提出了称之为切片符号执行的方法. 通过这种方法,用户可以通过指定某些范围的代码,将这段代码在分析的过程排除,从而将搜索过程集中在更重要的部分. 该文的重点在于解决切片过程中的副作用,如果能够结合自动分析程序热点会更加适用于实际程序的分析. Hristina Palikareva[10]提出了比较新旧程序补丁的不同版本生成用例的方法,即比对两个补丁代码的路径,当到达造成新老不同的分支点时,根据新的分支生成针对新补丁的方法,这样可以避免重复测试老的补丁代码. Katch[11]是基于 KLEE 开发的一款专门针对软件补丁的符号执行工具,主要方法是首先通过静态分析计算出测试用

例到达补丁代码的距离,随后选择距离最小的测试用例作为种子进行启发式的搜索. Edmund Wong 等人提出了基于文档辅助的建模方法 DASE[4],以及文档[12]中提到的基于程序功能标签切片的方法. 这两种方法都是从帮助文档、代码注释提取功能标签、文件格式等约束,从而对执行路径进行裁剪. 其中 DASE 方法以可执行和可链接类型的文件为目标,通过自然语言提取规则从帮助文档中提取对文件结构体的描述,比如格式文件中起始字节需要等于某些固定值. 但 DASE 方法并不能描述文件格式数据块之间的依赖而生成约束,该文也提到其构造的文件格式并不完整. 所以针对此方法的局限性,本文提出了基于文件格式数据块约束的符号执行分析方法,该方法以格式文件分析类应用程序为目标的,主要关注读取分析文件的功能路径,一方面能优化 DASE 中提出的约束提取方法,另一方面能够分析文件各部分的依赖关系以裁剪执行路径.

6　结束语

本文提出了一种基于文件格式的符号执行优化方案. 本文选择了 7 个常用文件处理软件作为测试对象,包括 Tcpdump、Readelf、Elfdump、File、Zlib、Flac、Imageinfo,解析的目标文件格式分别为传输协议包(TCP, Transmission Control Protocol),可执行和可链接文件,GNU 压缩文件(GZIP, GNUzip),标签图像文件(TIFF, Tag Image File Format)等. 实验结果表明 FFCBSE 相比于 KLEE 能够达到更高的代码覆盖率以及找到更多缺陷,相比于 DASE 方法也有一定程度提升. FFCBSE 发现了 13 个 KLEE 未发现的错误,同时指令覆盖率和分支覆盖率约有 10% ~225% 的提升.

后续的研究工作主要从以下两个方面开展:

(1)前文中提到,FFCBSE 使用了一个现有的格式文件作为输入,该文件只能是部分符合文件格式的要求. 比如在测试 readelf 软件中使用了一个在 ubuntu 平台生成的可执行文件,该文件可能并未满足 ELF 格式的所有特性,如何够构造一个满足规范的格式文件是后续的研究内容之一.

(2)考虑到大多数文件处理程序中关联操作只会涉及到两个文件,所以前文的算法描述中设计了读取两个文件的处理流程,但也不排除会涉及到三个及其以上数量的文件两两之间有关联. 如何有效的处理更多文件之间的交互情况也是后续的改进内容之一.

参考文献

[1] Wang M,Liang J,Chen Y,et al. SAFL:Increasing and accelerating testing coverage with symbolic execution and guided fuzzing[A]. Proceedings of the 40th International Conference on Software Engineering:Companion Proceeedings[C]. Gothenburg Sweden:IEEE Computer Society,2018. 61 - 64.

[2] Corteggiani N,Camurati G,Francillon A. Inception:system-wide security testing of real-world embedded systems software[A]. 27th USENIX Conference on Security Symposium[C]. Baltimore, MD: USENIX Association, 2018. 309 - 326.

[3] Mechtaev S, Nguyen M D, Noller Y, et al. Semantic program repair using a reference implementation[A]. Proceedings of the 40th International Conference on Software Engineering[C]. Gothenburg Sweden:IEEE Computer Society, 2018. 129 - 139.

[4] Wong E, Zhang L, Wang S, et al. DASE: Document-assisted symbolic execution for improving automated software testing[A]. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering[C]. Florence Italy: IEEE Computer Society,2015. 620 - 631.

[5] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[A]. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008[C]. San Diego, California: USENIX Association, 2008. 209 - 224.

[6] 杨超,郭云飞,扈红超,等. 基于符号执行的软件缓存侧信道脆弱性检测技术[J]. 电子学报,2019,47(6):1194 - 1200.
YANG Chao, GUO Yun-fei, et al. Cache-based side-channel vulnerability detection based on symbolic execution [J]. Acta Electronica Sinica, 2019, 47(6): 1194 - 1200. (in Chinese)

[7] 赵祖威,冯世宁,汤恩义,等. 一种符号执行制导的循环内界分析方法[J]. 电子学报,2017(11):16 - 26.
ZHAO Zu-wei, FENG Shi-ning, et al. A symbolic execution guided inner loop bound analysis[J]. Acta Electronica Sinica, 2017(11): 16 - 26. (in Chinese)

[8] 徐超,陈勇,葛红美,等. 基于符号执行的能耗错误检测方法[J]. 电子学报,2016,44(5):1040 - 1050.
XU Chao, CHEN Yong, GE Hong-mei, et al. Symbolic execution based energy bug detecting method [J]. Acta Electronica Sinica, 2016, 44(5): 1040 - 1050. (in Chinese)

[9] Trabish, David, Mattavelli, Andrea, Rinetzky, Noam, et al. Chopped symbolic execution[A]. Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings[C]. Gothenburg Sweden: IEEE Computer Society, 2018. 350 - 360.

[10] Palikareva H, Kuchta T, Cadar C. Shadow of a doubt: testing for divergences between software versions [A]. IEEE/ACM International Conference on Software Engineering[C]. Austin, TX: IEEE, 2016. 1181 - 1192.

[11] Marinescu P D, Cadar C. KATCH: high-coverage testing of software patches[A]. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering[C]. Saint Petersburg Russia: ACM, 2013. 235 – 245.

[12] 甘水滔,王林章,谢向辉,等. 一种基于程序功能标签切片的制导符号执行分析方法[J]. 软件学报,2019,30(11):3259 – 3280.

作者简介

汪孙律(通信作者) 男,1986 年 10 月出生,安徽安庆人,中国科学院软件研究所博士生,主要研究方向为软件安全.
E-mail: sunlv@nfs.iscas.ac.cn

李明树 男,1966 年 5 月出生,吉林德惠人,博士,中国科学院软件研究所研究员、博士生导师,主要究方向是:操作系统与基础软件,软硬件协同设计,以及软件工程方法和软件过程技术等.

杨秋松 男,1977 年 8 月出生,河北泊头人,博士,中国科学院软件研究所研究员、博士生导师,主要研究方向是:软件工程、形式化方法、模型检测、安全操作系统.