基于懒符号执行的软件脆弱性路径求解算法

秦晓军 周 林 陈左宁 甘水滔
(江南计算技术研究所 江苏 无锡 214083)

摘 要 为了解决软件测试中路径爆炸、新路径发现率低以及静态分析中虚报率高等问题,提出了动静态分析结合的脆弱性挖掘框架,并针对循环爆炸问题设计了基于懒符号执行的路径求解算法,该路径求解算法应用最短路径、条件约束集概率和可达路径数量 3 种静态信息制导符号执行,提高了路径选择的准确率,能较快地逼近脆弱点,并利用懒符号执行技术自动识别循环结构,通过推迟变量实例化等方法,有效地缓解了循环结构的路径组合爆炸问题,最终生成到达脆弱点集的数据包. 对 coreutils6.10 命令包的实验结果表明,与现有的方法 KLEE、Otter 和 SAGE 相比,该文提出的方法可以有效地对具有较多分支的程序进行分析,当测试程序越大其优势越明显.

关键词 软件脆弱性;静态分析;懒符号执行;条件约束集概率
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2015.02290

Software Vulnerable Trace's Solving Algorithm Based on Lazy Symbolic Execution

QIN Xiao-Jun ZHOU Lin CHEN Zuo-Ning GAN Shui-Tao
(*Jiang Nan Computer Technique Institute*, *Wuxi*, *Jiangsu* 214083)

Abstract To solve path explosion, low rate of new path's finding in the software testing and high rate of false alarm of static analysis, this paper proposes a vulnerability discovering architecture which combined dynamic analysis and static analysis, and design the trace solving algorithm based on lazy symbolic execution for the problem of loop explosion. This trace solving algorithm applies 3 key factors consisting of shortest path, constraint probability and reachable trace number to guide the symbolic execution which can reach the vulnerability faster with the more accuracy of trace choosing. Through the lazy symbolic execution to automatically identify the loop structure and delay the variables' concreting, we can fit the problem of trace combination explosion of loop structure efficiently and get the test cases which can reach the vulnerability sets. Our algorithm is also tested on coreutils6.10 and compared with KLEE, Otter and SAGE. The experiment result shows that our algorithm can analyze the program containing more branches effectively, and the larger testing program is, the more obvious advantage it has.

Keywords software vulnerability; static analysis; lazy symbolic execution; condition constraint sets probability

1 引 言

动态分析方法和静态分析方法是目前软件脆弱性检测技术的两种主要技术路线. 静态分析以脆弱性判定模型为指导,运用程序分析技术发现目标代码中的脆弱点,能够报告大部分脆弱点. 但由于不真正运行程序,静态分析方法难以准确计算出达到脆

收稿日期:2013-08-28;最终修改稿收到日期:2015-04-07. 本课题得到国家"八六三"高技术研究发展计划项目基金(2012AA7111043)资助.**秦晓军**,男,1975 年生,硕士,高级工程师,中国计算机学会(CCF)学生会员,主要研究方向为软件确保. E-mail: xjqin@163.com. **周 林**,男,1986 年生,硕士,工程师,主要研究方向为软件确保. **陈左宁**,女,1957 年生,硕士,高级工程师,中国工程院院士,主要研究领域为计算机软件与理论. **甘水滔**,男,1986 年生,硕士,工程师,主要研究方向为软件确保.

弱点的路径，误报率很高.另一方面，静态分析方法不具备脆弱点的自动验证能力，大部分情况下需要依赖于人工分析确认并构造触发数据，不但费时费力，而且并不能保证准确率.

动态分析方法的基本思路是利用测试用例动态执行程序中的每个分支，通过收集路径信息挖掘程序脆弱性，具有较高的准确率，目前主要分为模糊测试和污点分析/符号执行两种思路.动态分析方法的主要障碍在于测试效率不高，虽然研究人员对之进行了各种改进[1-4]，但动态分析方法始终面临如何提高执行路径覆盖率和解决路径爆炸等问题，所需时间复杂度和空间复杂度往往与被检测的程序代码量呈指数增长的关系，对于较大规模的软件显得力不从心.

动态测试和静态分析在很大程度上具有互补性.对前者而言，只有能覆盖脆弱性所在语句的测试输入才是有意义的，如能获取程序中潜在脆弱点的类型和位置，就能在不损害脆弱性发现能力的前提下，通过避免生成和执行无效的输入来降低测试开销.而静态分析方法虽然误报率高，但能提取程序中潜在脆弱语句的位置和类型信息，如果能有效地利用静态分析的结果来指导动态测试，将带来两个方面的优势：一方面，静态分析过程能提供目标脆弱点集和附加的信息，提高动态测试的针对性和有效性，另一方面，动态测试能更准确地验证脆弱点是否真正存在并有效定位，弥补静态测试方法虚报率过高的缺陷.

针对上述现状，本文设计了一种动静态分析结合的脆弱性挖掘框架，先通过静态分析获取疑似脆弱点集，再计算能到达疑似脆弱点集的可达数据包，如果有解，则确认了该脆弱点的可触发性，同时自动生成了可触发该脆弱点的输入数据，如图 1 所示.

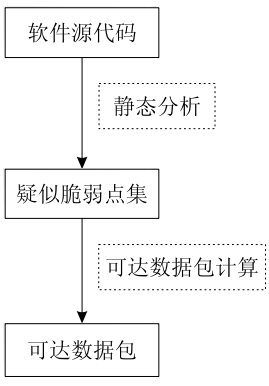

图 1　动静态分析结合的脆弱性挖掘框架

基于该框架，本文提出了基于脆弱点集制导的可达数据包计算模型 VTsolver，该模型结合动态符号执行和静态分析的优点，实现计算脆弱点可达数据包的自动化.

VTsolver 模型包含了本文提出的懒符号执行、约束条件集概率指导等方法，能够有效地缓解路径爆炸、虚报率过高等问题.其基本思想是：先通过静态分析获取疑似脆弱点集，并计算提取路径概率、路径长度和可达路径数量等指导性参数，然后利用可达数据包计算技术获得能到达脆弱点的数据包，在可达数据包计算过程中，之前提取的路径概率、路径长度和可达路径数量等信息用于反馈指导路径搜索.对符号执行过程中的循环组合爆炸问题，VTsolver 模型提出的懒符号执行的思想予以有效缓解.

VTsolver 模型在进行动态符号执行的同时搜集执行路径对应的路径约束条件，当确定到达脆弱点后，通过求取对应执行路径的路径条件，可以得到该执行路径上的脆弱点数据包.

本文第 2 节介绍目前国内外在脆弱点可达数据包计算技术方面的相关工作；第 3 节详细描述基于懒符号执行的路径求解算法；第 4 节通过实验将基于懒符号执行的路径求解算法与目前比较主流的符号执行工具 KLEE，Otter，SAGE 进行了性能比较；第 5 节对全文进行总结.

2　相关工作

基于制导的脆弱点可达数据包计算技术是最近几年才发展起来的软件脆弱性分析技术.早期研究人员在模糊测试中引入制导思想，用来排除无用路径，解决覆盖率问题[1,5-6].符号执行技术取得进步之后，制导检测思想应用到符号执行中用于计算能够到达脆弱点的数据包，以提高路径搜索的准确性.

2011 年崔展齐等人[7]提出了一种目标制导的混合执行测试方法，利用目标脆弱点的可达性静态信息来指导符号执行.同年 Ma 等人[8]提出了 3 种脆弱点可达数据包计算算法：最短距离符号执行(Shortest-Distance Symbolic Execution，SDSE)，调用链后向符号执行(Call-Chain-Backward Symbolic Execution，CCBSE)和混合调用链后向符号执行(Mixed Call-Chain-Backward Symbolic Execution，Mix-CCBSE).SDSE、CCBSE 和 Mix-CCBSE 这 3 种算法在路径选择策略时利用的都是最短路径优先思想，其中，SDSE 算法是利用当前节点和脆弱点之间的最短路径，CCBSE 利用的是当前节点和程序执行入口点之间的最短路径，Mix-CCBSE 利用的是两个当前节点

(初始的两个当前节点分别是程序入口和目标脆弱点)之间的最短路径.上述算法的主要问题在于,如果利用最短路径选择策略导向的后续搜索中约束集不可解或路径不可达,则符号执行引擎将浪费大量时间在无效的路径上.

2012 年 Chen 等人[9]利用动态符号执行得到的控制流图信息,采用带有限状态机的扩展程序行为模型(Extended Program Behavior model with Finite-State Machine controlled parameters,EPBFSM)指导模糊测试的执行路径.同年 Pak[10]做了类似工作,但其对非线性约束不能很好处理,不能准确产生非线性约束中符号变量的范围值,另外由于要处理模糊测试和符号执行的衔接,因此效率不高,内存消耗较大.

目前的符号执行方法还存在一个共同的问题,即遇到多重循环代码时,会遭遇严重的路径组合爆炸,并有很大可能漏报循环内的越界脆弱性,当前循环难题已作为符号执行研究中亟待解决的挑战性问题被提出,本文提出的懒符号执行方法较好地缓解了该问题.

3 VTsolver 模型

目标制导的符号执行方法主要解决的问题是:在给定脆弱点位置的条件下,如何找到一条可达路径触发给定脆弱点并得到相应的测试数据包.程序的执行可以用状态进行描述,脆弱点的可达性可以通过检查当前状态是否满足特定的性质来进行判断.

利用符号执行引擎,目标制导的符号执行技术能较好地解决脆弱点可达数据包的计算问题.符号执行引擎是 VTsolver 模型中的一个完备执行系统,只要是它能执行的路径都等价于数据包可达.符号执行引擎同时使用具体参数和符号参数驱动程序运行,符号参数可以对基本数据类型,字符串等进行抽象.当符号执行引擎执行时遇到符号化的条件语句时,将调用可满足性模理论(Satisfiability Modulo Theories,SMT)求解器 STP(Simple Theorem Prover)对搜集到的约束集进行求解,判断分支是否可达,以及有多少分支是可达的,如果有多条路径可达,则需要把这些状态保留下来,用于路径的遍历.

VTsolver 模型应用最短路径、条件约束集概率和可达路径数量 3 种关键静态信息制导符号执行,提高了路径选择的准确性,能较快的到达脆弱点,并得到相应的数据包.特别地,针对目前循环体符号执行时间开销过大的问题,VTsolver 提出了一种懒符号执行方法,在"必要时"才对循环体进行符号执行,极大减少了无效的符号执行运算开销.

懒符号执行方法主要包含基于懒符号执行的前向路径求解算法(Frontward trace solve algorithm based on Lazy Symbolic Execution,FLSE),基于懒符号执行的后向路径求解算法(Backward trace solve algorithm based on Lazy Symbolic Execution,BLSE),基于懒符号执行的混合路径求解算法(Hybrid trace solve algorithm based on Lazy Symbolic Execution,HLSE).

3.1 基本定义

定义 1(节点距离). 过程间控制流图(Interprocedural Control Flow Graph,ICFG)中从当前结点到脆弱点路径内的边数目,称为当前结点和脆弱点之间的距离.

定义 2(节点可达性). 对于 ICFG 中任意两节点 n_1 与 n_k,满足可达条件当且仅当:对于 i, $0<i<k$, $n_i \in D\text{-}Pred(n_{i+1})$,存在一条路径 $path=\langle n_1, n_2, \cdots, n_k \rangle$,其中 $D\text{-}Pred(n_{i+1})$ 表示节点 n_{i+1} 的直接前驱集合.

定义 3(条件约束集概率). 条件约束集概率定义为条件约束集的解集数量与解集空间基数的比率.

3.2 模型框架

VTsolver 模型的基本流程如图 2 所示.

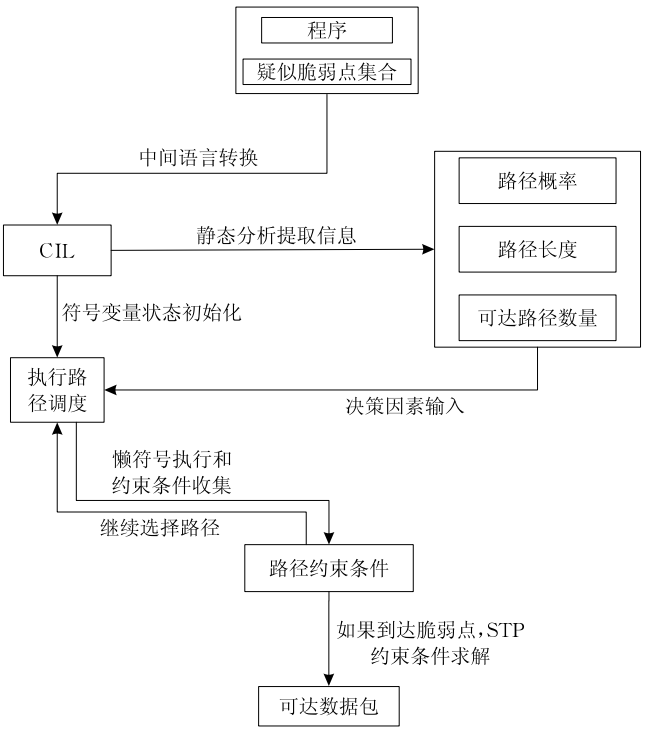

图 2 VTsolver 模型

不管是高级语言还是二进制语言，都会隐含各种类型和内存访问等信息，有些情况下存在二义性，如 C 语言中指针与数组的定义，形式上可能完全相同，但具有不同的内存访问模式. 二义性的存在使代码不够直观，不便于自动化分析，需要转换成中间语言，并且转换后的中间语言不能丢失源语言中的类型等重要信息. 本文使用 CIL 中间语言，CIL 是一种源到源（可逆）的中间语言，比较适合源代码分析.

第 1 步. 先将程序代码转换成中间语言代码 CIL 表示形式，转换前疑似脆弱点集在程序代码中已经被标记；

第 2 步. 在转换后的 CIL 中间语言表示的基础上提取条件约束集概率、路径长度和路径数量等静态信息；

第 3 步. 初始化懒符号执行调度器，并输入条件约束集概率，路径长度和路径数量 3 种调度因素，调度器开始工作并根据调度策略选择分支路径供懒符号执行引擎执行；

第 4 步. 懒符号执行引擎执行路径并搜集路径对应的约束条件；

第 5 步. 利用符号约束求解器求解约束条件得到可达数据包.

VTsolver 的功能模块划分如图 3 所示，整个模型可以分为前端和后端两个部分，前端主要由脆弱点集标记模块和 CIL 中间语言转换模块组成. 后端主要由路径调度（路径策略选择）模块、懒符号执行模块（Lazy Symbolic Execution，LSE）、静态分析模块和 STP 符号约束条件求解模块组成.

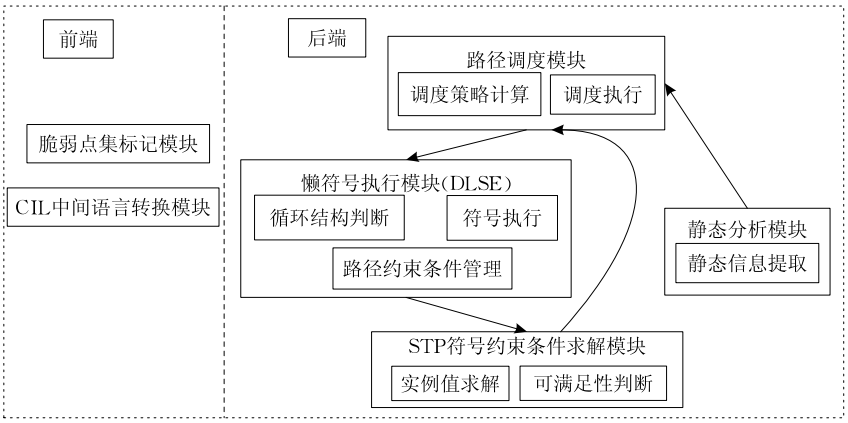

图 3　VTsolver 模型的功能模块划分

前端的脆弱点集标记模块将脆弱点集在程序中标记出来作为后端静态分析和懒符号执行等模块的基本输入. CIL 中间语言转换模块将目标源代码转换为 CIL 中间语言，后端执行均基于 CIL 中间语言进行，这样不仅提高了模型的扩展能力，并且能够实现对多种程序设计语言的支持. CIL 左值表达式通过〈内存地址，对象类型〉二元属性组来进行表示，而高级语言中只用〈内存地址〉表示左值表达式，导致同一个语句会产生二义性，CIL 中间语言可以消除这种二义性.

VTsolver 模型后端的路径调度模块主要综合 3 种决策因素对路径进行决策，并把下一条要执行的路径信息传递给懒符号执行模块.

懒符号执行模块是 VTsolver 模型的核心模块之一，主要包括 3 个功能：

（1）循环结构判断. 对需要进行懒符号执行的语句或过程进行判断，由于循环结构尤其是多层循环是影响符号执行效率的重要因素[11]，为了简化其中的判断规则，懒符号执行只对涉及符号实例化的循环结构进行处理；

（2）符号执行. 对确定的语句进行符号执行，其中包括实例化的具体程序执行和符号化的程序模拟执行；

（3）路径约束条件管理. 通过将新收集的符号约束加入到已有的符号约束集，将得到与输入相关的新变量以及关系表达式，同时将懒符号执行中自主选择的新路径反馈给路径调度模块.

FLSE 算法、BLSE 算法和 HLSE 算法在以上所描述的功能模块基础上进行实现，其不同之处体现在符号执行的方向和策略上，FLSE 算法是从程序的执行入口点开始搜索脆弱点，最后搜索到整条执行路径. BLSE 算法是从脆弱点位置开始搜索直到达程序执行入口，逆向得到整条执行路径，HLSE 算法是从程序入口点和脆弱点同时开始并行搜索，最后把前段和后段相连得到整条执行路径.

由于懒符号执行算法和条件约束集概率计算是

FLSE 算法、BLSE 算法和 HLSE 算法的重要子算法,下面先介绍懒符号执行算法和条件约束集概率计算,然后再介绍 FLSE 算法、BLSE 算法和 HLSE 算法.

3.3 懒符号执行算法

懒符号执行算法的基本思想是当符号执行引擎遇到循环分支或多重循环分支时,将循环变量符号化,推迟符号变量的实例化,即:先对循环体进行符号执行,再根据分支条件与循环变量、符号变量的关系回溯执行循环结构,否则按原有序列执行.将循环计数变量符号化,而不是按照通常的做法实例化循环计数变量并执行循环结构体,这种处理方法能够避免大量循环分支路径的产生,另一方面,通过对循环结构体的考察能更加精确地求解出循环计数变量的可解范围,从而指导符号执行的路径选择,使得执行更加有效.

在进行懒符号执行之前,相关静态信息已经计算完成,比如可达性信息,因此在描述算法时不再赘述静态分析方面的行为.懒符号执行过程如算法 1 和算法 2.

算法 1. 懒符号执行算法*LazySymbolicExecute*.

输入:l:表示第 l 行语句
δ:表示路径条件约束集
m:表示关于符号变量的映射
输出:δ:新路径条件约束集

```
1.  while ~branch(l) do //判定是否是分支
2.      m←m∪m(v,e) /*其中 v,e 为 l 中的变量,v 为被赋值对象,e 为被 v 数据依赖的变量*/
3.      l←next(l)    //读下一条语句
4.  endwhile
5.  if ~isloop(l) then    //判断是否是循环语句
6.      c←m(cond(l))
7.      if SAT(δ∧c) then /*SAT 为约束求解判定器*/
8.          /*如有解则符号执行语句 l,否则符号执行下一条语句*/
            LazySymbolicExecute(target(l),δ∧c,m)
9.      else if SAT(δ∧~c) then
10.         LazySymbolicExecute(next(l), δ∧~c, m)
11.     ENDIF
12. ELSE
13.     //是循环语句,调用懒符号执行引擎函数
        LazySymbolicEngine(l,δ,m)
14.     LazySymbolicExecute(next(l),δ∧~c,m)
15. ENDIF
```

算法 2. 循环体执行算法*LazySymbolicEngine*.

输入:l:表示第 l 行语句
δ:表示路径条件约束集
m:表示关于符号变量的映射
输出:δ:新路径条件约束集

```
1. symbolic(&i)   //将循环变量 i 符号化
2. if isloop_end(l) then //判断是否循环结构结束
3.     RETURN δ //返回更新后的约束集
4. ELSE
5.     LazySymbolicExecute(target(l),δ∧c,m)
6. ENDIF
```

l 表示第 l 行语句,δ 表示路径条件约束集合,m 表示关于符号变量的映射集合,$branch(l)$判断 l 是否是分支语句,对顺序执行语句 l:$v=e$,$next(l)$表示下一条执行语句,$m(v,e)$表示新发现的映射关系.对分支语句 l,$target(l)$表示跳转的目标语句,$isloop(l)$判断 l 是否是循环语句,$cond(l)$表示语句 l 中的约束条件,c 为临时变量,保存 $cond(l)$的值,*LazySymbolicEngine* 对循环结构进行懒符号执行.$isloop_end(l)$表示判断语句 l 是否为循环体结束.

算法 1 中第 1~4 行判断当前执行的语句是否为分支语句并更新符号变量映射,第 5~12 行判断分支语句是否为循环语句,如果是循环语句,则进行懒符号执行;否则搜集约束条件,并利用 SAT 求解器来判断路径是否可行,第 13 行表示搜集关于循环变量的可解约束条件.第 14 行执行循环结构.

算法 2 表示对循环结构体进行符号执行.第 1 行对循环变量 i 进行符号化,第 2 行判断是否是循环结构体结束,如果是循环结构体结束,则返回包含关于 i 的约束集 δ,第 5 行搜集结构体约束信息.

懒符号执行可以有效解决执行循环的盲目性问题.例如考察以下一段代码:

```
1.  symbolic(&m); //变量 m 符号化
2.  symbolic(&n); //变量 n 符号化
3.  for(i=0; i<MAXLOOPNUM1; i++)
4.  {
5.      for(j=0; j<MAXLOOPNUM2; j++)
6.      {
7.          statement_m(m,i,j);//变量 m 的赋值依赖 i,j
8.          statement_n(n,i,j); //变量 n 的赋值依赖 i,j
9.          …
10.         if(condition(m, n)) //条件分支
11.             g(m, n);
12.         …
13.     }
14. }
```

第 1、2 行代码标记变量 m、n 为符号变量，第 7、8 行代码说明 m、n 的值由 i、j 决定，对符号变量 m、n 赋值实例化后，调用函数 $g(m,n)$. 如函数 $g(m,n)$ 可到达脆弱点，一般的符号执行算法将 i，j 实例化并进入到循环结构体进行执行，当遇到第 10 行时，如果不满足条件则返回继续尝试新的循环值. 懒符号执行算法则先符号执行 $g(m,n)$，确定可到达脆弱点的 m、n 的约束条件，然后根据第 7、8 行列出 m、n 和 i、j 的函数关系确定 i、j 的约束条件. 这样避免了对 i、j 的进行盲目的符号执行，极大地提高了符号执行的效率.

动态符号执行在一定程度上解决了缺乏测试用例集的问题，但由于该方法从程序本身出发，未将目标脆弱性的先验知识作为指导，导致生成和执行了大量无效（不能覆盖脆弱性语句）的测试输入，浪费了时间和计算资源. 另外，符号执行在执行多重循环程序时，会遭遇严重的路径组合爆炸问题，懒符号执行通过推迟循环符号变量的实例化，先对循环结构内的语句进行符号执行，反推可到达脆弱点的条件，然后根据可达性条件有选择地进行符号执行，较好地克服了这个问题.

3.4　约束条件集概率

在进行符号执行时，除收集约束条件，还需要收集约束条件集概率，约束条件集概率描述的是约束条件集可解的概率有多大，约束条件集和执行路径是一一对应的，因此将其作为路径选择的一个因素可以使得所选择的路径更加合理. 本文通过算法 $probSymCalculate(l,\psi,m,p)$ 来求取约束条件集概率，其中 l 表示代码所在行数，ψ 表示当前收集到的条件约束集，m 表示关于符号变量的映射集合，p 表示 ψ 约束集概率. 算法 $probSymCalculate(l,\psi,m,p)$ 的具体描述见算法 3.

算法 3.　约束条件集概率求解算法 $probSymCalculate(l,\psi,m,p)$.

输入：l：表示第 l 行语句

　　ψ：表示路径条件约束集

　　m：表示关于符号变量的映射

　　p：中间约束条件集概率

输出：p：约束条件集概率

1. while $\sim branch(l)$ do
2. 　$m \leftarrow m(v,e)$
3. 　$l \leftarrow next(l)$ //读下一条语句
4. endwhile
5. $c \leftarrow m(cond(l))$
6. $\psi' \leftarrow relation(\psi,c)$
7. $p_c \leftarrow prob(\psi' \wedge c)/prob(\psi')$
8. $probSymCalculate(target(l),\psi \wedge c,m,\ p * p_c)$ //为真分支
9. $probSymCalculate(next(l),\psi \wedge \sim c,m,p*(1-p_c))$ //为假分支

$relation(\psi,c)$ 表示抽取 ψ 中与 c 相关的条件约束 ψ'，设 ψ 中与 c 独立的条件约束为 $\bar{\psi}$，由于 ψ' 与 $\bar{\psi}$ 独立，c 与 $\bar{\psi}$ 独立，因此 $c \wedge \psi'$ 与 $\bar{\psi}$ 独立，对分支语句，需要提取约束条件，并更新约束集 ψ，约束集 $\psi \wedge c$ 的概率可以通过条件概率来进行计算：

$$Prob(\psi \wedge c) = Prob(\psi) * Prob(c|\psi)$$

因为 $\psi = \psi' \wedge \bar{\psi}$

$$\Rightarrow$$

$$Prob(\psi \wedge c) = Prob(\psi) Prob(c|(\psi' \wedge \psi)) = \frac{Prob(\psi) Prob(c \wedge (\psi' \wedge \bar{\psi}))}{Prob(\psi' \wedge \bar{\psi})}$$

因为 $Prob(\psi' \wedge \bar{\psi}) = Prob(\psi') Prob(\bar{\psi})$，并且

$$Prob(c \wedge (\psi' \wedge \bar{\psi})) = Prob(c \wedge \psi') Prob(\bar{\psi})$$

$$\Rightarrow$$

$$Prob(\psi \wedge c) = \frac{Prob(\psi) Prob(c \wedge \psi') Prob(\bar{\psi})}{Prob(\psi') Prob(\bar{\psi})} = \frac{Prob(\psi) Prob(\psi' \wedge c)}{Prob(\psi')} = p \cdot p_c$$

约束集 $\psi \wedge \sim c$ 的概率可以通过条件概率来进行计算：

$$Prob(\psi \wedge \sim c) = Prob(\psi) Prob(\sim c|\psi) = Prob(\psi)(1 - Prob(c|\psi)) = p \cdot (1 - p_c)$$

本文使用 LattE[①] 对约束概率进行求解，LattE 接受“=”和“≤”表达式，“≥”和“>”表达式可以通过乘以 −1 转换为“<”和“≤”表达式，“<”可以通过减去一个常数来转换成“≤”表达式，不直接支持“≠”表达式.

约束条件集概率通过对单个约束条件概率的计算获得，求取单个约束条件概率的算法见算法 4.

算法 4.　单个约束条件概率求解算法.

$probConditons(\psi \vdash \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n)$.

输入：ψ_i：约束条件

输出：p：新路径条件约束集

1. $ConSet \leftarrow \{\psi_1, \psi_2, \cdots, \psi_n\}$;
2. $Vars \leftarrow \{var | \exists\, con \in ConSet \wedge var \in con\}$;
3. $neqConSet \leftarrow \{con | con \in ConSet \wedge con \vdash \neq relation\}$;

① UC Davis, Mathematics. Latte integrale. http://www.math.ucdavis.edu/~latte

4. $lgeConSet \leftarrow Conset - neqConSet$;

5. $unSet \leftarrow \{lgeConSet \wedge var = exp \mid var \neq exp \in neqConSet\}$;

6. $num \leftarrow num_{\wedge}(\wedge lgeConSet) - num_{\vee}(\vee unConSet)$;

7. $p \leftarrow pnum \Big/ \prod_{var \in Vars} \sharp var$

为了让 LattE 支持“≠”表达式,首先将约束条件集 $ConSet$ 分解为两个集合,一个是不含“≠”的表达式集合 $lgeConSet$,另一个是含“≠”的表达式集合 $neqConSet$. $unSet$ 集合为满足 $lgeConSet$ 条件约束,但是不满足 $neqConSet$ 条件约束的约束集.

考虑到 $unSet$ 集合中元素之间的相关性问题,根据容斥原理,约束条件集解的数量 $num_{\vee}(\vee unSet)$ 计算公式如下:

$$num_{\vee}(\vee unSet) = \sum_{\theta \in unSet^{*}} (-1)^{\sharp\theta - 1} num_{\wedge}(\wedge_{\theta_i \in \theta} \theta_i),$$

其中 $unSet^{*}$ 表示 $unSet$ 的幂集,$\sharp\theta$ 表示集合 θ 的元素数量,$num_{\wedge}(\wedge_{\theta_i \in \theta}\theta_i)$ 表示 $\wedge_{\theta_i \in \theta}\theta_i$ 解的数量.

3.5 基于懒符号执行的路径求解算法

3.5.1 基于懒符号执行的前向路径求解算法

SDSE,Klee 等系统利用优先执行最短路径策略来进行脆弱点位置的前向搜索,执行的策略只以当前信息即最短路径作为唯一决策因素,没有考虑约束集概率和后续路径数量等关键因素,决策因素过于简单,因此当最短路径对应的约束集无解时,算法必须回溯到上一个分支节点,否则浪费太多时间在无解的路径上.

基于懒符号执行的前向路径求解算法(FLSE)是一种综合考虑了最短路径、约束概率和可达路径数量 3 种关键因素对下一条执行路径进行选择,并对选择路径进行懒符号执行的目标制导脆弱点可达数据包计算法. FLSE 算法在决策出下一条执行分支后,会先判断下一条执行分支是否是循环分支,如果不是循环分支则进行正常的符号执行,如果是循环分支则提取循环条件,将循环控制变量符号化,对循环结构体进行符号执行,对循环结构体进行切片和数据流分析,变量的依赖关系,活性分析,循环分支之间的独立性等性质,得到关于循环控制变量的约束信息. FLSE 算法描述如算法 5.

算法 5. FLSE 算法 *FLSESymbolicExecute*.

输入:*vulset*:脆弱点集

输出:*vulcase*:到达脆弱点集的数据包

1. *initial*(*tasklist*,*vulset*,*main*)
2. *probSymCalculate*(l,ψ,m,p)
3. while (*tasklist*!=NULL)
4. l=*select* (*tasklist*)
5. *LazySymbolicExecute* (l,δ,m)
6. endwhile

其中 *initial*(*tasklist*,*vulset*,*main*)表示对符号执行引擎进行初始化,将初始的执行状态添加到 *tasklist* 中,*probSymCalculate*(l,ψ,m,p)计算约束条件集概率,*select*(*tasklist*)选取一个任务状态 l,*LazySymbolicExecute*(l,δ,m)对 l 进行懒符号执行.

FLSE 算法一方面通过提前确定和求解多个执行分支,减少了不必要的回溯,不需要在一个过程内反复的修正约束条件集,解决了符号变量受父节点控制时所带来的过程内约束条件集无法修正的问题. FLSE 算法通过提前求解若干节点,利用后续节点的分支信息,有效地提高了符号执行的效率. 另一方面,一次符号执行跨越多个节点,得到更多关于循环符号变量的信息,减少了循环分支数量,从而减少了影子内存的数量,有效地缓解了符号执行的状态数量爆炸问题.

3.5.2 基于懒符号执行的后向路径求解算法

CCBSE 算法与 SDSE 算法类似,利用优先执行最短路径策略来进行脆弱点位置的后向搜索,执行的策略也只以当前信息即最短路径作为唯一决策因素,没有考虑约束集的求解概率和前续路径数量等关键因素,决策因素过于简单. 而且 CCBSE 算法每步只执行一个分支,没有充分利用后续分支进行综合决策.

基于懒符号执行的后向路径求解算法(BLSE)从脆弱点所在函数开始,通过一次多步逆向追溯的方法搜索整个执行路径,BLSE 对下一条执行路径进行选择时综合考虑了最短路径、约束集概率和可达路径数量 3 种关键因素,有效地提高了路径搜索效率. 其算法描述如算法 6.

算法 6. BLSE 算法 *BLSESymbolicExecute*.

输入:*vulset*:脆弱点集

输出:*vulcase*:到达脆弱点集的数据包

1. *initial*(*tasklist*,*vulset*,*vulfun*)
2. *probSymCalculate*(l,ψ,m,p)
3. while (*tasklist*!=NULL) do
4. l=*select*(*tasklist*)
5. *LazySymbolicExecute*(l,δ,m)
6. *next_l*=*CFGBackForward*(l)
7. *add*(*tasklist*,*next_l*)
8. endwhile

其中 *initial*(*tasklist*,*vulset*,*vulfun*)表示对符号执

行引擎进行初始化，这里与 FLSE 算法有所不同，FLSE 算法是从程序入口点开始进行符号执行的，BLSE 算法是从目标脆弱点开始的，$vulfun$ 是脆弱点所在函数. $probSymCalculate(l,\psi,m,p)$ 计算约束条件集概率，$select(tasklist)$ 选取一个任务状态 l，$LazySymbolicExecute(l,\delta,m)$ 对 l 进行懒符号执行，$CFGBackForward(l)$ 选取下次要的执行路径，并通过 $add(tasklist,next\_l)$ 加入到任务列表 $tasklist$ 中.

BLSE 算法有两个关键的步骤：(1) 在过程间控制流图中找到调用当前函数的调用函数集，并跟设定的策略选择一个较好的调用函数作为下一个符号执行函数；(2) 在选定好的调用函数中使用 FLSE 算法，找到一条较好路径到达目标函数.

第一个关键步骤中从调用函数集中选取一个较好的调用函数优先执行，其选取策略包含两个决策因素：调用函数与被测程序入口之间的距离和调用函数与被测程序入口之间路径的数量. 如图 4 所示，BLSE 算法先确定脆弱点所在函数 $vulfun$，并利用 FLSE 算法，找到一条从 $vulfun$ 函数入口到脆弱点的执行路径 $(p_1,p_2,\cdots,p_k)$，并搜集相应的路径约束条件 s_1，然后在 $vulfun$ 的直接前驱集 $D\text{-}Pred(vulfun)$ 中，利用决策函数 $decision(Pathlen,Pathnum)$ 求出直接前驱集 $D\text{-}Pred(vulfun)$ 中各元素的权重值，并优先执行权重最大的直接前驱结点，其中 $Pathlen$ 表示直接前驱结点到被测函数执行入口之间的距离，$Pathnum$ 表示直接前驱结点到被测函数执行入口之间的路径数量. 节点距离 $Pathlen$ 和路径数量

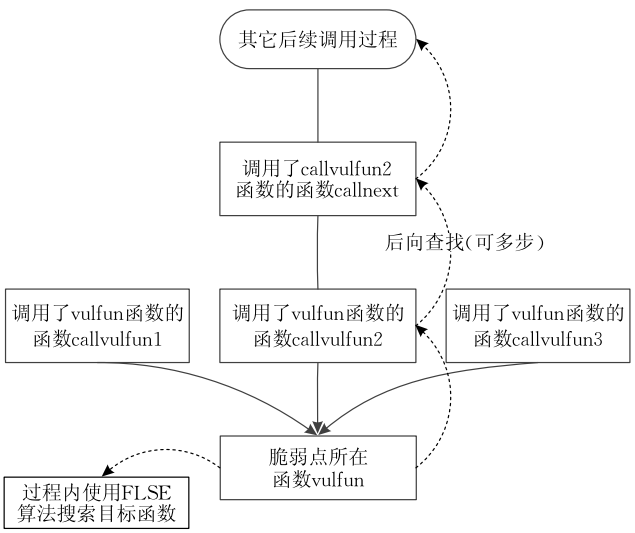

图 4　BLSE 执行示意图

$Pathnum$ 是预先可以计算的，因此直接前驱权重值 $decision(Pathlen,Pathnum)$ 也是可以预先计算的，从而不会因为直接前驱权重值 $decision(Pathlen,Pathnum)$ 的计算而影响整个算法的执行效率，这里没有考虑约束条件概率因素，主要是因为在后向执行路径搜索时，约束条件概率的求取比较复杂，而且效果也不一定好.

在决策出下一条执行分支 $callvulfun2$ 后，利用 BLSE 算法搜索 $callvulfun2$ 函数内从 $callvulfun2$ 函数入口到调用 $vulfun$ 函数之间的执行路径 $(p_{k+1},p_{k+2},\cdots,p_{k2})$，并搜集相应的条件约束 s_2，同时并不会直接对所得到的条件约束进行求解，而是预先考察下个分支中的分支和分支条件情况，分析下个分支函数入口与调用 $callvulfun2$ 语句之间的可达性条件，将其加入到约束集中，并进行综合求解，如果调用 $callvulfun2$ 语句总是可达的，则不需要加入约束集中，简化条件约束求解，提高求解效率.

3.5.3　基于懒符号执行的混合路径求解算法

FLSE 算法和 BLSE 算法都有本身不可避免的不足，FLSE 算法适用于前后搜集的路径约束条件信息，基于懒符号执行的混合路径求解算法（HLSE）结合了 FLSE 算法和 BLSE 算法的优点，同时从程序入口和脆弱点两个节点同时出发进行条件路径搜索，这样在进行路径选择时，得到的信息更多，选择出来的分支也更加合理. 具体算法描述如算法 7.

算法 7.　HLSE 算法 $HLSESymbolicExecute$.

输入：$vulset$：脆弱点集

输出：$vulcase$：到达脆弱点集的数据包

1. $initial(ftasklist,vulset,main)$
2. $initial(btasklist,vulset,vulfun)$
3. $probSymCalculate(l,\psi,m,p)$
4. $FLSESymbolicExecute(ftasklist)$
5. $BLSESymbolicExecute(btasklist)$

其中，$ftasklist$ 和 $btasklist$ 分别为 FLSE 和 BLSE 的状态集，$initial(ftasklist,vulset,main)$ 表示对 $ftasklist$ 进行初始化，对 $main$ 参数和全局变量的符号化，生成初始状态. $initial(btasklist,vulset,vulfun)$ 表示对 $btasklist$ 进行初始化，对脆弱点调用函数的参数和全局变量进行符号化并产生初始状态. $probSymCalculate(l,\psi,m,p)$ 计算约束条件集概率，$FLSESymbolicExecute(ftasklist)$ 和 $BLSE\text{-}SymbolicExecute(btasklist)$ 分别进行前向懒符号执行和后向懒符号执行.

4 实验与分析

本实验的测试对象为以上的 3 个例子和 BUSYBOX 1.11.2,测试结果如表 1 所示.

表 1 3 种懒符号执行算法与 KLEE[12],SAGE[13],Otter[8] 性能比较 (单位:s)

测试包	算法					
	FLSE	BLSE	HLSE	KLEE	SAGE	Otter
kill	12.88	17.15	15.56	65.66	2012.64	16.94
setuidgid	8.47	258.51	16.18	26.74	3892.12	21.63
tr	8.82	21.70	16.27	21.42	38.43	18.20
od	9.17	34.79	39.44	54.32	191.94	49.14
chown	24.29	105.21	112.80	255.71	596.12	114.10
ls	923.27	966.54	130.87	275.95	1899.93	151.55
均值	164.48	233.98	55.19	116.63	1438.53	61.93

以对 od 命令包的测试为例分析,FLSE,BLSE 算法的测试效率最高,HLSE 算法的测试效率一般,因为这个程序较为简单,程序分支较少,因此在前向约束条件和后向约束条件联合求解上消耗的时间就显的较为突出. KLEE 算法的测试效率较好,KLEE 使用的是 Round-Robin RP(Radom Path)算法来计算路径末端与脆弱点之前的距离,同时兼顾了覆盖率和路径执行的准确率. Otter 采用最短路径的路径选择策略,测试效率较好. SAGE 注重提高覆盖率,覆盖每个执行分支,通过块覆盖(block coverage)进行启发式选择状态执行,因此把太多的时间浪费在不能到达脆弱点的程序分支上.

以对 ls 命令包的测试为例分析, HLSE 测试效率最高,FLSE,BLSE,KLEE,Otter 的效率次之,SAGE 的效率最差,因为 ls 程序中无效的分支太多,因此 SAGE 浪费在这些分支上的时间显得特别突出.

从对 BUSYBOX 中 ls,chown,tr,od,kill 和 setuidgid共 6 个命令的测试结果数据可以看出,SAGE 的性能表现是相对最差的,FLSE 在对其中 5 个命令的测试过程中表现优异,BLSE 在对其中 4 个命令的测试性能较好,但两个算法在对 ls 命令的测试中出现性能突变,表现不够稳定,HLSE 的综合效率是最高的,并且随着被测试代码规模的增大,HLSE 算法的优势越来越明显. 测试性能比较见图 5.

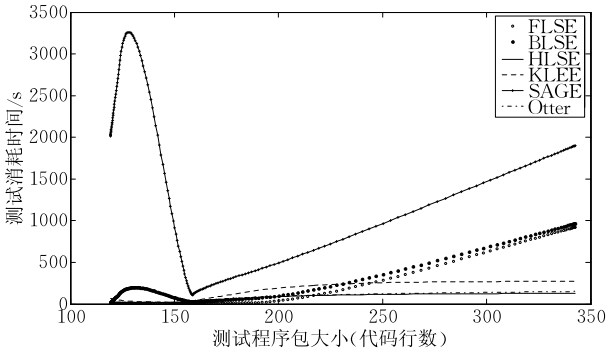

图 5 算法性能测试比较

图 5 中纵坐标表示对程序进行测试花费的时间,单位为秒,横坐标表示被测程序的代码规模,单位为行数. 从图 5 中可以看出 SAGE 在对 setuidgid 程序包进行测试时效果非常差,因为 setuidgid 中循环比其它测试数据包多,且不能达到脆弱点的无用路径也较多,SAGE 为了提高覆盖率,在遍历这些路径时浪费了太多的时间.

最后,VTsolver 在针对软件 coreutil6.10 的脆弱点测试中,快速的生成了 9 个漏洞可达数据包,其中包括 2 个(序号 2 和序号 7)未公开漏洞,详细情况见表 2.

表 2 懒符号执行对 coreutil6.10 软件包测试的生成结果

序号	测试目标	脆弱性类别	公开编号	生成数据包
1	chown	指针误用	* * * *	"a.a" "-"
2	od	栈溢出	* * * *	"//" "//" "B"
3	ls	整数溢出	CVE-2003-0854	"-w" "1073741828"
4	tr	栈溢出	* * * *	"[a-z"
5	kill	指针误用	* * * *	"-l" "a"
6	pr	指针误用	* * * *	"-e" "file2.txt" file2: "\b\b\b\b\b\b\b\t"
7	mkdir	栈溢出	* * * *	"-Z" "\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x00" "\x32\x20\x20\x20\x20\x20\x20\x20\x20\x20\x00"
8	mknod	竞争条件	* * * *	"-m" "400" "foo"
9	dir	整数溢出	CVE-2003-0854	"-w" "1073741828"

表 2 中列出了 9 个命令包的测试结果,有 3 个指针误用脆弱点,3 个栈溢出脆弱点,两个整数溢出脆弱点和一个竞争条件脆弱点,其中 dir 和 ls 命令的脆弱点已经收录在 CVE 中.

5 总　结

脆弱点可达数据包计算是指在进行程序分析时，对于给定的代码脆弱点位置，找到一条可达路径触发给定脆弱点和得到相应可达数据包. 程序的执行可以用状态进行描述，脆弱点的可达性可以通过检查当前状态是否满足特定的性质来进行判断. 脆弱点可达数据包计算技术应用很广泛，比如一个静态分析的脆弱性测试工具找到了一些脆弱点，但是并不能确定是不是真正的错误，于是需要测试数据包来验证这些脆弱点. 在进行软件开发的时候可能会遇到一些断言(assert)报告，但是没有相应的测试数据包，如果开发者对代码不熟悉或者代码量比较大，要找到触发断言的原因是非常困难的，因此需要测试数据包帮助开发者定位错误.

符号执行技术非常适合解决脆弱点可达数据包计算问题，符号执行引擎是一个完备的执行系统，只要是它能执行的路径都是可达的. 符号执行引擎调用程序运行，同时使用具体参数和符号参数. 符号参数可以对基本数据类型，字符串等进行抽象. 当符号执行引擎执行时遇到符号化的条件语句时，将调用SMT 求解器 STP 对符号执行引擎搜集到的约束集进行求解，判断分支是否可达，有多少分支是可达的，如果有多条路径可达，则需要把这些状态保留下来，用于路径的遍历.

本文提出了基于懒符号执行的前向路径求解算法(FLSE)，基于懒符号执行的后向路径求解算法(BLSE)和基于懒符号执行的混合路径求解算法(HLSE)，有效地提高了符号执行引擎的执行效率.

FLSE 算法综合考虑了最短路径、约束概率、可达路径数量 3 种关键因素，对下一条执行路径进行选择，有效地提高了路径搜索效率. 另外 FLSE 算法在决策出下一条执行分支后，并不会直接对所得到的条件约束进行求解，而是预先考察下个分支中的分支和分支条件情况，如果满足设定的提前策略，能够确定再下个执行分支，则把这个分支对应的条件加入当前约束条件集，同样可以对后面的执行分支实施提前策略，直到遇到不能满足提前策略的结点.

BLSE 算法从脆弱点所在函数开始，通过一次多步逆向追溯的方法搜索整个执行路径，BLSE 综合考虑了最短路径、约束概率、可达路径数量 3 种关键因素，对下一条执行路径进行选择，以提高路径搜索效率. BLSE 算法有两个关键的步骤：(1) 在过程间控制流图中找到调用当前函数的调用函数集，并跟设定的策略选择一个较好的调用函数作为下一个符号执行函数；(2) 在选定好的调用函数中使用 FLSE 算法，找到一条较好路径到达目标函数.

HLSE 算法结合了 FLSE 算法和 BLSE 算法的优点，同时从程序入口和脆弱点两个节点同时出发进行条件路径搜索，这样在进行路径选择时，得到的信息更多，选择出来的分支也更加科学.

参　考　文　献

[1] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing//Proceedings of the International Conference on Software Engineering. Vancouver, Canada, 2009: 474-484

[2] Li Jia-Jing, Wang Tie-Lei, Wei Tao, et al. Polynomial time path-sensitive taint analysis method. Chinese Journal of Computers, 2009, 32(9): 1845-1855(in Chinese)
(李佳静，王铁磊，韦韬等. 一种多项式时间的路径敏感的污点分析方法. 计算机学报，2009，32(9)：1845-1855)

[3] Chen Kai, Feng Deng-Guo. Black-box testing based on colorful taint analysis. Journal of Chinese Science: Information Science, 2011, 41(5): 526-540(in Chinese)
(陈恺，冯登国. 基于彩色污点传播的黑盒测试方法. 中国科学：信息科学，2011，41(5)：526-540)

[4] Zhu Guan-Miao, Zeng Fan-Ping, Yuan Yuan, Wu Fei. Blackbox fuzzing testing based on taint check. Journal of Chinese Computer Systems, 2012, 33(8): 1736-1739 (in Chinese)
(朱贯森，曾凡平，袁园，武飞. 基于污点跟踪的黑盒 fuzzing 测试. 小型微型计算机系统，2012，33(8)：1736-1739)

[5] Godefroid P, Klarlund N, Sen K. Dart: Directed automated random testing//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Chicago, USA, 2005: 213-223

[6] Babic D, Martignoni L, McCamant S, Song D. Statically-directed dynamic automated test generation//Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. Toronto, Canada, 2011: 285-296

[7] Cui Zhan-Qi, Wang Lin-Zhang, Li Xuan-Dong. Target-directed concolic testing. Chinese Journal of Computers, 2011, 34(6): 953-965(in Chinese)
(崔展齐，王林章，李宣东. 一种目标制导的混合执行测试方法. 计算机学报，2011，34(6)：953-965)

[8] Ma K-K, Phang K Y, Foster J S, Hicks M. Directed symbolic execution//Proceedings of the 18th International Static Analysis Symposium (SAS). Venice, Italy, 2011: 365-380

[9] Chen Zhe, Guo Shize, Fu Damao. A directed fuzzing based on the dynamic symbolic execution and extended program behavior model//Proceedings of the Instrumentation, Measurement, Computer, Communication and Control (IMCCC'12). Harbin, China, 2012: 1641-1644

[10] Pak B S. Hybrid Fuzz Testing: Discovering Software Bugs Via Fuzzing and Symbolic Execution[M. S. dissertation]. Carnegie Mellon University, Pittsburgh, USA, 2012

[11] Saxena P, Poosankam P, McCamant S. Loop-extended symbolic execution on binary programs//Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. Chicago, USA, 2009: 512-528

[12] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs//Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. San Diego, USA, 2008: 623-635

[13] Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing//Proceedings of the 15th Annual Network and Distributed System Security Symposium. San Diego, USA, 2008: 320-332

QIN Xiao-Jun, born in 1975, M. S., senior engineer. His main research interest is software assurance.

ZHOU Lin, born in 1986, M. S., engineer. His main research interest is software assurance.

CHEN Zuo-Ning, born in 1957, M. S., senior engineer, academician of Chinese Academy of Engineering. Her main research interest is computer software and theory.

GAN Shui-Tao, born in 1986, M. S., engineer. His main research interest is software assurance.

Background

Software analysis is a hot issue in the area of information security at present. The vulnerability mining is one of the most important parts of software analysis. There are many methods to mine the software's defects, for example fuzz testing, taint analysis, static analysis and symbolic execution, etc. The fuzz testing is applied widely; however, it has the problem of path explosion, low rate of new path finding and the low coverage. It also has to take much time to analyze the program because it is lack of a smart guide. The taint analysis has the same of problem as fuzz testing. Though many researchers try their efforts to improve them, no essential improvement is get. Static program analyses are used by many developers to test their programs because they are effective in finding some trivial bugs that can be caught by the rules that define security violations with very small resource. However, they are limited in that the performance is only good as the rules and have high rate of false alarm.

Symbolic execution is another technique that has recently gotten the attention of security researchers. In contrast to fuzzing, symbolic execution tests a program by treating the program's input as symbols and interpreting the program over these symbolic inputs. In theory, symbolic execution is guaranteed to be effective in achieving high code coverage, yet this generally requires exponential resource which is not practical for many real-world programs.

Our goal is to find more bugs faster than traditional approaches. In order to accomplish this goal, firstly we find the vulnerability sets using static analysis, and then we need to obtain line reachable case in reasonable resource bound (e.g. computing power and time). Getting line reachable case implies both breadth and depth in exploration of the program. Although we may not achieve the best code coverage or speed, we aim to find the sweet spot in cost-effective way to gain more real vulnerabilities than the fuzzer and the symbolic executor. In this thesis, we present our attempt to attain the best of both worlds by combining symbolic execution with fuzzing in a novel manner. Our vulnerability mining model first uses static analysis to discover vulnerabilities in the program. After collecting vulnerabilities under a user-configurable resource constraint, we use the trace solving algorithm to get the line reachable suits of the vulnerabilities automatically. This algorithm based on lazy symbolic execution applies three key factors of shortest path, constraint probability and reachable trace number to guide the symbolic execution. It can raise the correctness rate of path selection and reach the vulnerability faster. To tackle the problem of trace combination explosion of loop structure, the lazy symbolic execution is applied which can automatically identify the loop structure and delay the variables' concreting. The experiment result shows that our algorithm can analyze the program of more branches effectively.