一种C程序内存访问缺陷自动化检测方法研究

杨 飏[1] 张焕国[1,2,3] 王后珍[1]

(空天信息安全与可信计算教育部重点实验室 武汉 430072)[1]

(武汉大学计算机学院 武汉 430079)[2] (武汉大学软件工程国家重点实验室 武汉 430072)[3]

摘 要 符号执行是目前较为行之有效的软件缺陷自动化检测方法,计算代价昂贵与程序执行路径爆炸是两个影响其性能的关键问题。提出了一种针对C语言程序内存访问缺陷的符号执行检测方法,该方法可通过自动化构造的测试用例发现程序内部的内存访问缺陷,如缓冲区溢出、跨界访问和指针异常等。使用符号跟踪缓冲区长度的方法,一方面减少了符号变量的数量,另一方面由此精确抽象C语言库中字符串操作函数的行为,解决了符号执行过程间函数调用的步进问题;使用动态切片的方法,裁减路径探索过程中的冗余路径,从而解决在程序内部路径搜索时发生的路径爆炸问题。实验表明,提供的检测方法不但可行,而且验证代价较小,具有较强的实用性。

关键词 静态分析,符号执行,程序切片,约束求解

中图法分类号 TP309 **文献标识码** A

Full-automatic Detection of Memory Safety Violations for C Programs

YANG Yang[1] ZHANG Huan-guo[1,2,3] WANG Hou-zhen[1]

(The Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education, Wuhan 430072, China)[1]

(Department of Computer Science, Wuhan University, Wuhan 430079, China)[2]

(State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)[3]

Abstract Symbolic execution is an effective and automatic bug-finding method. But symbolic execution is limited in practice by the computation cost and path explosion. We presented a tool for full-automatic detection and generating inputs that lead to memory safety violations in C programs, including buffer overflow, buffer overrun and pointer dereference error. In order to reduce the amount of symbol variable, we used the symbol variable to track the length of buffer and C string. The use of symbolic buffer length makes it possible to compactly summarize the behavior of standard buffer manipulation functions. While resolving the problem of path explosion, we introduced the dynamic slicing methods to prune the redundant paths. It's shown by the experiments that our method presented in this paper not only is feasible but also has little cost.

Keywords Static analysis, Symbolic execution, Program slicing, Constraint solving

随着国家和社会对软件系统的依赖程序日益增长,复杂软件系统的正确、安全和可靠对相关基础设施与应用是至关重要的。对于个人,软件系统中的错误和缺陷可能导致黑客窃密、隐私人侵和财产损失等安全威胁。

验证与测试是提高软件可信程序的重要方法。当前软件验证的实践主要采用模型检测[1]、程序静态分析[2]和形式化程序验证[3]3种方式。国内外学者在不同领域进行了大量相关研究。总体说来,软件验证存在公理化系统建立较为复杂、程序正确属性与断言定义困难、分析结果存在大量误报和漏报等问题。测试是另一种在工业界广泛使用提高软件生产质量保障的重要技术,因其简单有效得到了迅速发展,主要目的是尽可能地减少程序的错误。统计表明,在典型的软件开发项目中,软件测试工作量往往占软件开发总工作量的50%左右。开销巨大的主要原因在于测试的自动化程序不高、测试用例的生成与挑选需要手工处理;另一方面,随着软件技术的不断发展,不断发现各种新的软件错误或缺陷的类型,这也给软件测试技术提出了新的问题与挑战。

符号执行是一种介于程序验证与测试之间的一种方法。符号执行由J. C. King[4]于1976年提出,通过程序的输入数据将符号引入到程序的执行语义中,程序的执行通过符号来模拟。在此基础上,张健[5,6]、Engler[7,8]等人使用符号执行的方法自动地检测软件中的错误和缺陷。其原理是沿程序执行图跟踪收集符号输入变量的约束集,采用搜索策略遍历程序内部执行分支。在任意执行路径上,若检测到软件发生错误,如程序崩溃、断言错误和安全属性违背等,则由约束求解器生成满足约束条件的具体测试用例。该测试用例保证程序沿符

到稿日期:2009-07-08 返修日期:2009-09-25 本文受国家863高技术研究发展计划项目基金(2007AA01Z411,2008AA01Z404),国家自然科学基金(90718005,90718006)资助。

杨 飏(1980—),男,博士生,主要研究方向为可信计算与信息安全,E-mail:little_mud2004@yahoo.com.cn;张焕国(1945—),男,教授,博士生导师,主要研究方向为信息安全与可信计算;王后珍(1982—),男,博士生,主要研究方向为信息安全等。

号执行选择的路径真实执行，从而发现真正引起错误的软件缺陷。

符号执行是目前较为行之有效的软件缺陷自动化检测方法，但在处理大规模软件系统时会遇到以下两个问题：一是符号变量计算代价昂贵，如一个 1024 字节的测试缓冲区需要用 1024 个符号变量表示，并且与外界函数调用之间的交互处理较为复杂；二是由于需要搜索程序内的所有可行路径，待检测的执行路径数量爆炸，这也从另一个角度导致约束求解的任务加剧。

本文提出的符号执行检测系统针对软件缺陷中最为常见、危害最大的内存访问缺陷展开研究，使用基于反例的约束求解方法进行检测。在符号执行系统设计中，对其所面临的两个关键问题进行优化。

针对第一个问题，引入了“符号抽象长度”的思想，用符号跟踪输入缓冲区的长度，而并非缓冲区的内容，从而尽量减少符号变量的数量；精确地抽象 C 语言库中字符串操作函数的行为，从而解决过程间调用时的符号步进执行的问题，这也在一定程度上减少了搜索路径数量。

针对第二个问题，引入了“上下文动态切片”的概念。符号执行的上下文描述了基于某种搜索策略的符号执行，到达路径中的某程序点时，程序中所有变量所满足的条件。本文认为，如果程序的执行到达某个程序点时，与在此之前的程序点有相同的符号执行上下文，那么会产生相同的后续结果，并且两个程序点之间的执行路径可以被裁减。在程序执行前期，对路径的动态裁减，将对后面的执行路径带来指数级别的约简。

第一种方法，可以看作是“水平”方向的符号空间精确；第二种方法，则可以看作是“垂直”方向的执行路径裁减。两种方法的综合使用，可以解决符号执行中代价昂贵、路径爆炸的问题。实验证明，本文的方法不仅可行，而且验证代价相对较小，因此具有良好的实用性。

本文第 1 节介绍符号执行的基本原理—控制流图和执行模型；第 2 节介绍符号执行抽象和内存访问缺陷检测技术；第 3 节分析符号执行动态切片；第 4 节分介绍实验系统框架与实验结果；第 5 节介绍相关工作；最后给出总结。

1 控制流图与执行模型

基于路径的测试是以程序控制流图（Control Flow Graph，简称为 CFG）为基础的。

定义 1 控制流图 CFG 是一个有向图 $G=\{X,\mathcal{L},op,E\}$，其中 X 为变量集合，子集 $x_0\subseteq x$ 为输入向量；$\mathcal{L}$ 代表程序基本块节点的集合，$l_0\in\mathcal{L}$ 为特定的开始结点；函数 op 标识 $l\in\mathcal{L}$ 中块节点的基本操作 $\{halt|abort|x:=e|\text{if}(x)\ \text{then}\ l'\ \text{else}\ l''\}$，其中 $halt$ 为程序正常中止；$abort$ 为程序错误退出；赋值语句 $x:=e$，$x\in X$ 且 e 是 X 上的运算表达式；条件语句 if (x) then l' else l''，$x\in X$ 且 $l',l''\in\mathcal{L}$；$E\subseteq\mathcal{L}*\mathcal{L}$ 表两个节点之间的转移，$N(l)$ 为 l 唯一的邻居节点。

程序的内部运行过程可以用流图中的路径来刻画。控制流图中的一条执行路径 A 可以用节点序列表示为 $A=l_1l_2l_3\cdots l_m$，也可以用边的序列表示为 $A=e_1e_2e_3\cdots e_m$，其中 m 表示路径 A 的长度，并且 $e_i=\langle l_i,l_{i+1}\rangle$。

从控制流图的角度，对程序执行给出如下定义。

定义 2 在程序 P 中，C 为条件语句集合，A 为赋值语句集合。程序执行 ω 为一个有限序列 $Execs:=(A\cup C)*(abort|halt)$。

执行模型是对计算机在执行程序时行为的形式化描述。对于一个程序的具体语义，可以理解为利用内部存储空间为程序变量赋值和计算。对于符号执行来说，通过物理存储空间 M 和新加的符号空间 S，分别为表达式中的具体或符号变量赋值。符号与具体数交织执行（concolic execution）的程序执行语义如图 1 所示。

```
Input:Program P=(X,L,op,E)
Result:Path constration ζ
S:=S+[m ↦ input]
ζ:=emptyStack;l=l0;
while op(l)≠(abort,halt) do
  switch op(l) do
    case(m←e):
      S=S+[m ↦μ(e)];
      M=M+[m ↦M(e)];
      l=N(l);
    end
    case(if (x) then l'else l'')
      if M(x)=0 then
        ζ:=push(μ(x)=0,ζ);
        l=l'';
      else
        ζ:=push (μ(x)≠0,ζ)
        l=l';
    end
  end
end
if(S==abort) then
  raise an exception
else //S==halt
  returen ζ;
```

图 1 符号执行程序的执行语义

在此，使用 $M':=M+[m\mapsto v]$ 表示 $M'(m)=v$ 的物理空间更新，使用 $S':=S+[m\mapsto v]$ 表示符号空间的更新。路径约束 ζ 用于沿执行路径收集符号变量的约束条件，可以形象化理解为栈操作。对于任意执行路径而言，对路径约束 ζ 的求解保证沿该路径的具体执行。

通过分析可以看出，符号执行是将程序内部的隐式执行逻辑通过符号的约束条件显式化，并由生成输入来取代消费输入。执行路径收集到的约束将输入进行等价类划分，同一类的符号输入将执行相同的执行路径。

2 符号执行中的抽象

2.1 长度符号化

在 C 语言程序代码中，跨界访问、缓冲区溢出等内存安全违背之所以非常普遍，一方面在于数组和指针解引用的操作并不自动进行边界检查，需要程序员自行检查；另一个重要的原因是许多标准 C 库支持的字符串操作函数，如 *strcpy*()，*strcat*()，*sprintf*()，*gets*() 都是不安全的。如图 2 的代码所示，当输入字符串为 1024 字节长度时，会导致 off-by-one 的内存访问错误。

```
01  void Function1(char * src){
02  unsigned int length;
03  int dest[1024];
04  length=strlen(src);
05  if(length>1024)
06  return;
07  strcpy(dest,src);
08  dest[length+1]=0;}
```

图 2　C 语言代码片断

针对该错误，使用 Fuzzing 黑盒自动化测试技术，需要构造至少 1024 个随机测试用例进行测试。使用传统符号执行测试，在 *strlen*，*strcpy* 函数源代码可用情况下，需要构造至少包括 1024 个符号变量的缓冲区作为测试向量，并且步进过程间调用函数跟踪各符号变量的约束条件。

本文通过观察发现，此类安全问题主要与缓冲区或字符串的长度相关，而与其内容并无本质关联。为此，仅针对字符串与缓冲区长度进行符号跟踪，而对输入内容随机构造。

定义 3　对于符号缓冲区，使用〈*baseadd*，*endadd*，ζ，η〉四元组的内存结点来表示。其中 *baseadd* 与 *endadd* 为符号缓冲区在内存中的具体映射地址，ζ 为符号缓冲区的分配长度，η 为缓冲区内存储的符号字符串长度。

在具体实现中，可以使用 splay 树数据结构来存储和排序程序中的内存结点，并不区分物理地址是由堆或栈上分配。

2.2　字符串操作函数建模

具备了长度符号化表示条件后，可进一步对 C 语言标准库中的字符串操作函数行为进行抽象建模，一方面可以解决过程间调用时的路径爆炸的问题；另一方面通过加入断言，跟踪检查内存访问缺陷，如缓冲区长度是否大于所包含的字符串的长度。

相关的内存结点操作函数与 C 语言库字符串操作函数抽象模型如图 3 所示。

```
//memory node allocation and free
  malloc(s)→createnode(baseadd,endadd,μ(s))
  * p→Mnode=find(p)
  free(p)→Mnode=find(p)
          delete(Mnode)
//C operation function
  l=strlen(s)→Mnode=find(s)
              l=μ(Mnode,η)
  strpcy(dest,src)→Mnode_d=find(dest);Mnode_s=find(src)
                   Mnode_d.η=Mnode_s.η
                   assert(Mnode_d.η<=Mnode_d.η)
  strcat(dest,src)→Mnode_d=find(dest);Mnode_s=find(src)
                   Mnode_d.η=Mnode_d.η+Mnode_s.η
                   assert(Mnode_d.η<=Mnode_d.ζ)
```

图 3　内存操作与字符串操作函数建模

2.3　反例约束求解

在使用符号变量实际追踪 C 语言中的内存访问时，本文采用反例约束求解的方法：将导致内存溢出的约束条件加入到路径约束集中，如果有可满足解，则存在相应的安全漏洞。

以图 2 的代码为例，可以随机构造符号输入缓冲区开始测试。在 04 行变量 *length* 直接得到符号长度 η。当 $\eta>1024$ 第一轮执行结束，未发现安全问题，第二轮执行。当代码执行到 07 行时，*dest* 的符号向量长度为 η，进行断言与路径约束联合检查 $assert(\eta\leqslant 1024)\wedge\neg(\eta>1024)$，未发现断言违背，程序继续执行。当代码执行到 08 行时，根据反例约束求解，检查 $\neg(\eta>1024)\wedge(\eta+1)>1024$，发现可满足解为 $\eta=1024$，即输入缓冲区为 1024 个字节时程序内部 off-by-one 内存访问错误。

3　上下文动态切片

3.1　符号执行上下文

在符号执行时，当输入的符号出现在分支语句的条件判断中，为了搜索所有可能性，需要分别执行两条路径，并在每条分支增加合适的约束条件。具体算法如图 4 所示。

```
//rule for if(e) s1 else s2
conditional_rule(e,s1,s2)
 concrete      symbolic
   if e     |  if for k()==child
     s1     |    push_constraint(e=ture)
{           |    s1
   else     |  else
     s2     |    push_constrainte(e=false)
            |    s2
}
```

图 4　条件语句的符号执行

条件分支语句路径的增长会给后继的路径搜索带来指数级别的影响，从而带来路径爆炸问题。“上下文动态切片”的思想在于，一旦程序执行到某程序点，如果与之前的程序点具有相同的程序状态（路径约束集），则认为后续的执行将得到相同的结果，从而该分支片段可以从后续的路径中裁减。

在实际处理中，由于在条件语句的两条分支选取不同的约束条件，从而导致出现约束命中的概率并不高。为此，本文将上下文的概念进行扩展，从而满足更广泛的应用。

定义 4　程序执行到某一程序点，如果与之前的程序点具有相同的程序状态，或者程序状态中的不同变量没有后续的读操作，则认为该分支片段可以从后续路径中裁减。

在此，将有后续读操作的符号变量称为存活变量（live symbol），将只包含存活变量的程序状态称为精确状态（Refined State）。

以图 5 为例，当程序执行到语句 02 时，首先，沿 *true* 分支执行，并在 03 语句增加约束条件 $x=1024$，继续执行 05 语句。因为在 02 语句后一直未对变量 x 有读操作，所以存活变量和精确状态都为空集，程序继续执行后继语句，完成一轮测试。

	State	live symbol	Refined State
01 x=sym_input(0;	{x=*}	{x}	{x=*}
02 if(x==1024)	{x=*}	{}	{}
03　printf(“true”);	{x=1024}	{}	{}
04 else printf(“false”);	{x≠1024}	{}	{}
05 …	{x=1024},{x≠1024}	{}	{}

图 5　C 语言片段与上下文

接下来，沿 false 分支执行，并在 04 语句增加约束条件 $x\neq 1024$。当执行到语句 05 时，因为也未对变量 x 有读操作，存活变量和精确状态都为空集，从而与 true 分支的状态产生碰

撞。由此认定当前分支片段是可裁减的，中止符号执行，由约束求解器产生具体测试用例，完成当前路径测试。后继路径的测试由当前程序点开始。

3.2 Look-Forward(LF)搜索

在动态切片中，在程序执行点确定存活变量(live symbol)是关键问题。存活变量确定得越精确，精确状态就可以得到更多的命中，可裁减的路径分支也就越多。

本文使用沿分支搜索(look forward)策略，对程序点之后的所有路径进行深度遍历，并执行存活变量分析，从而决定当前程序点路径是否继续扩展或裁减。具体算法如图 6 所示。

```
procedure SEARCH(node,Φ,H)
input:node,path constraint Φ,set of symbol H
output:update Tests
case node of
  ε→/ * end node * /
    try
      Sp:=SOLVE(Φ);Tests:=Tests+{Sp}
      H:={}
    with unsat→();
    end try
  ite(cond,inode,tnode)→
    if(reachForeward(inode,H)) then
      U=Search(inode,Φ∧cond)
      if H∩U≠∅
        then GoForwardBranch()
        else CutCurrent()
    if(reachForeward(tnode,H)) then
      U=Search(tnode,Φ∧cond)
      if H∩U≠∅
        then GoForwardBranch()
        else CutCurrent()
end case
```

图 6 存活变量搜索算法

对于沿分支搜索逻辑算法，有如下性质：

性质 1(完备性) 使用沿分支搜索逻辑的目的在于，根据程序状态丢弃冗余的分支，并且对程序节点的覆盖率没有影响。

性质 2(指数约减) 使用动态切片后，达到程序全覆盖需要更少的执行路径，并且在前端的裁减会给后端带来指数级别的影响。

4 实验评估

基于以上原理，本文实现了一个 C 语言缺陷自动化验证工具原型，如图 7 所示。本验证工具分为预处理过程中、决策过程和检测过程 3 部分。在预处理过程中，将源代码转换到中间语言形式使用的是 CIL 工具[9]，目的是在选定的程序执行路径上恢复程序的控制结构和与数据依赖关系，并进行插桩操作；在程序插桩时，对 C 语言的字符串操作函数进行抽象建模。在决策过程，对缓冲区与变量长度进行符号跟踪，收集通路上的约束条件，同时检测内存访问缺陷；在路径选择时，使用“动态切片”方法裁减程序语义无影响的分支。在检测过程中，利用约束求解器对路径约束集进行求解使用的是 STP 工具[10]，并对实际通路对其块覆盖率进行计算统计。

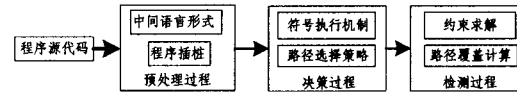

图 7 检测系统框架

本文利用两个示例程序，将上述缺陷验证工具与 EXE 系统[8]进行比较，后者是较为活跃且成熟实用的符号执行系统。对比的结果如表 1 所列。第一个程序 Example1 是图 1 所示的示例程序，第二个验证程序 Simple.c 来自于文献[8]，程序第 12 行存在一个缓冲区溢出错误。表中 Name 指 C 语言程序的名字；LOC 是指 C 程序的行数(未预处理前)；Num 为重复执行的总次数；Time 是重复符号执行程序所使用的时间，上面一组为本文的缺陷验证工具所花费的时间；Per 为与 EXE 系统相比节省时间与花费时间的比例。

表 1 实验结果比较

Name	Loc	Num	Time	Per
Example1	8	100	17s 21s	19%
Simple	24	70	28s 43s	35%

实践证明，本文提出的方法不仅可以有效地发现程序中安全缺陷，而且可以提高故障检测效率，达到了良好的实验效果。

5 相关工作

目前，有许多方法用于检测 C 语言程序中的内存安全属性违背。

静态分析。与符号执行相对比，静态分析可以运行在程序的所有路径上，并且不需要任何的测试用例。但是对于依赖准确的值信息进行推理的能力很差，从而产生大量的误报和漏报。Wanger[11]将缓冲区跨界访问(buffer overrun)看作整数边界分析问题，使用数据流不敏感分析字符串操作时的错误，该方法速度较快但不精确。

符号测试。使用符号执行进行执行路径搜索，目前主要的工具系统有 DART[12]，CUTE[13]和 EXE[8]。本文的一个贡献是使用符号跟踪缓冲区的长度而非内容，大大缓解了符号执行的工作负担。

路径搜索。对于符号执行中的路径爆炸的关键问题，各符号执行系统使用了多种方法，如使用启发式引导路径搜索、在函数调用时缓存函数摘要、或使用符号执行与随机测试相结合的方法。相对比，本文的动态切片方法更具有实用性。

结束语 本文提出了一种 C 语言程序内存访问缺陷自动化检测方法，方法综合使用了符号执行、符号抽象、动态切片、基于反例的约束求解等多种方法和技术，从而取得了较好的实验效果。

参 考 文 献

[1] Brat G, Havelund K, Park S, et al. Model checking programs [C]// IEEE International Conference on Automated Software Engineering. March 2000:25-37

[2] Xie Y, Kiken A. Saturn: A SAT-based tool for bug detection[C]// Proceeding of the Conference on Computer-Aided Verification. 2005:13-18

(下转第 185 页)

gen 所提供的 auction. dtd 中的内容模型。在这里我们不考虑上下文相关性，因此可以参照 auction. dtd 的内容模型来检验 XSDInfer 结果的正确性。

表 5　表达能力对比

元素(次数) \ 内容模型	1. auction. dtd 2. XSDInfer 3. XStruct
item (543)	1. a b c d e f g+ h 2. (a b c d e f)(g)⟨1,unbounded⟩(h) 3. (a b c d e f(g)⟨1,unbounded⟩h)+(a b c d e f g h)
person (637)	1. (a b c? d? e? f? g? h?) 2. (a b)(c? d? e? f? g? h?)⟨0,1⟩ 3. 47 个字符的表达式
address (304)	1. a b c d? e 2. a b c d? e 3. a b c((d e)+e)
open_auction (300)	1. a b? c* d e? f g h i j k 2. (a)(b)⟨0,1⟩(c)⟨0,unbouned⟩(d e? f g h i j k) 3. 94 个字符的表达式
profile (312)	1. a* b? c? d e? 2. a⟨0,unbounded⟩b⟨0,1⟩c⟨0,1⟩(d e?) 3. 31 个字符的表达式

从表 5 可以看出，XSDInfer 能够得到比较准确的结果，并在简洁性和可读性方面都优于 XStruct。

结束语　本文提出了一种基于内容模型图的 XSD 提取方法 XSDInfer，实验结果表明，与同类方法相比，XSDInfer 具有更好的可扩展性和表达能力。

在未来的工作中，我们将考虑对命名空间的支持。另外，虽然 tTRANS 规则给出了实际中常见情况下的处理方法，但是在时间和空间代价允许的前提下，我们仍然可以考虑对各种情况进行更加细致的划分，从而得到更精确的结果。

参 考 文 献

[1] Sun. JAXB[EB/OL]. http://java. sun. com/webservices/jaxb

[2] Bex G J, Neven F, Vansummeren S. Inferring XML schema definitions from XML data[C]// Proceedings of the 33rd International Conference on Very Large Data Bases. Vienna, Austria: VLDB Endowment, 2007: 998-1009

[3] Hegewald J, Naumann F, Weis M. XStruct: efficient schema extraction from multiple and large XML documents[C]// Proceedings of 22nd International Conference on Data Engineering Workshops. Atlanta, GA, USA, 2006: 81

[4] Altova XMLSpy[EB/OL]. http://www. altova. com/products/xmlspy/xmlspy. html

[5] Stylus Studio[EB/OL]. http://www. stylusstudio. com/

[6] Trang C J. Multi-format schema converter based on RELAX NG [EB/OL]. Thai Open Source Software Center Ltd, 2003. http://www. thaiopensource. com/relaxng/trang. html

[7] Garofalakis M, Gionis A, Rastogi R, et al. XTRACT: learning document type descriptors from XML document collections[J]. Data Mining and Knowledge Discovery, 2003, 7(1): 23-56

[8] Min J, Ahn J, Chung C. Efficient extraction of schemas for XML documents[J]. Information Processing Letters, 2003, 85(1): 7-12

[9] Bex G, et al. Inference of concise DTDs from XML data[C]// Proceedings of the 32nd International Conference on Very Large Data Bases. Seoul, Korea: VLDB Endowment, 2006: 115-126

[10] Martens W, Neven F, Schwentick T. Simple off the shelf abstractions for XML Schema[J]. ACM SIGMOD Record, 2007, 36(3): 15-22

[11] Thompson H, et al. XML schema part 1: Structures[S]. W3C Recommendation, 2001

[12] XMark[EB/OL]. http://monetdb. cwi. nl/xml/

[13] Miklau G, Suciu D. XML data repository[EB/OL]. http://www. cs. washington. edu/research/xmldatasets/

(上接第 158 页)

[3] Feng X, Ni Z, Shao Z, et al. An open framework for foundational proof-carrying code[C]// Proceeding of ACM SIGPLAN International Workshop on Types in Language Design and Implementation. 2007: 67-78

[4] King J C. Symbolic Execution and Program Testing[J]. Communications of the ACM, 1976, 19(7): 385-394

[5] Zhang Jian. Symbolic execution of program paths involving pointer and structure variables[C]// 4th International Conference on Quality Software(QSIC'04). 2004: 87-92

[6] Xu Z, Zhang J. A test data generation tool for unit testing of C programs[C]// International Conference on Quality Software (QSIC'06). 2006: 63-74

[7] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs[C]// USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008). San Diego, December 2008: 33-45

[8] Cadar C, Dunbar D. EXE: Automatically generating inputs of death[J]. ACM Transactions on Information and System Security, 2008, 12(2): 26-43

[9] Necula G C, McPeak S, Rahul S, et al. CIL: Intermediate language and tools for analysis and transformation of C programs [C]// International Conference on Compiler Construction. March 2002: 67-83

[10] Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays[C]// Proceeding of the 19th International Conference on Computer Aided Verification. 2007: 87-96

[11] Wanger D, Foster J S, Brewer E A, et al. A first step towards automated detection of buffer overrun vulnerabilities[C]// 7th Network and Distributed System Security Symposium. California, USA, 2000: 23-45

[12] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[C]// Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 2005: 213-223

[13] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C[C]// Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, USA, 2005: 263-272