基于分支混淆算法的符号执行技术

过辰楷 姬秀娟 许 静

(南开大学信息技术科学学院 天津 300071) (南京大学计算机软件新技术国家重点实验室 南京 210093)

摘 要 符号执行是静态分析中的一项常用技术,数组元素混淆问题是限制符号执行本身性能的关键因素之一。通过分析数组混淆实质,提出了一种分支混淆算法,利用边混淆边符号执行的策略,可以处理较为复杂的数组问题。该策略使用实时的约束求解,及时地剪除不可达的混淆分支。结合符号执行和约束求解技术,开发了基于分支混淆算法的工具原型 ASym。初步实验表明,利用分支混淆算法可以处理具有分支结构的数组混淆问题,避免延迟替换出现的数组语义误差,且在很大程度上缩减了分支数量,提高执行效率。

关键词 符号执行,软件测试,数组混淆,约束求解

中图法分类号 TP311 **文献标识码** A

Symbolic Execution Based on Branch Confusion Algorithm

GUO Chen-kai JI Xiu-juan XU Jing

(College of Information Technical Science, Nankai University, Tianjin 300071, China)

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract Symbolic execution is a common static analysis technology. Issue of array element confusion is one of the key factors limiting symbolic execution performance itself. Through analysis to array confusion essence, branch confusion algorithm was proposed. With the strategy that manages confusion algorithm and symbolic execution in the same time, some complex array problems were solved. Using the real time method of constraint solving, infeasible confusion branches were cut in time. Combining with symbolic execution and constraint solving, the prototypical tool ASym was developed, which was based on improved confusion algorithm. Primary experiments show that it can solve the confusion problem in branch structure and avoid array semantic error in delay replacement. Meanwhile, extensional branches are dramatically reduced and efficiency is improved.

Keywords Symbolic execution, Software testing, Array confusion, Constraint solving

1 引言

符号执行是一种对路径敏感的静态分析方法,由 King J C 于 1976 年提出[1]。其基本思想是以符号值代替具体的数值作为程序的输入,从而得出一系列符号变量表达式,通过对该表达式的约束求解,使其成为生成测试用例的依据。由于静态测试在效率上优于动态测试,符号执行技术已经成为软件测试研究的热门。一批动静结合的符号执行工具,如 DART[2]、CUTE[3]、EXE[4] 应运而生。

然而符号执行也存在一些先天的缺陷,特别是对于数组和复杂数据结构没有很好的处理办法[5]。这些结构语义灵活,无法根据单一的符号替换策略得出最终的符号结果表达式。下面的程序段可以说明这一点。

```
ElementType symbolArray(ArrayType []a, ElementType b)
  {
1) ElementType i=0;
2) a[0]=i;
3) a[b]=i;
4) return a[0];
  }
```

数组 a 的两个操作 $a[0]=i$ 和 $a[b]=i$ 虽然都是赋值,但含义差别巨大。赋值语句 $a[0]=i$,由于 $a[0]$ 数组索引是常量,其语义固定,因此可以直接使用符号值替换它;然而对于赋值语句 $a[b]=i$,由于其下标不固定,因此情况就会复杂得多。假设 a 是一个长度为 3 的数组,因为不知道将 i 值赋给 $a[0]$,$a[1]$ 还是 $a[2]$,所以存在歧义,符号执行无法正常进行下去。

针对符号执行中的数组元素混淆问题,已有学者从不同角度加以处理,然而也存在一些局限。文献[6]使用"懒替换策略",当遇到复杂数组结构时就使用一类新的变量来代替。然而,若一段程序中出现变量索引数组较多时,该策略会延迟执行大量的数组变量,削弱了符号执行的力度,大量语句得不到符号执行。当以复杂数组结构作为条件约束时,由于对复杂数组没有进行符号执行,因此该条件约束仍然存在数组混

到稿日期:2011-10-20 返修日期:2011-12-01 本文受南京大学计算机软件新技术国家重点实验室开放课题(KFKT2010B22),天津市科技攻关项目(08ZCKFGX01100)资助。

过辰楷(1988—),男,硕士,主要研究方向为软件测试、信息安全,E-mail:chenkai.guo@163.com;姬秀娟(1977—),女,博士,主要研究方向为软件测试、软件分析;许 静(1967—),女,教授,博士生导师,主要研究方向为软件测试、软件工程、信息安全检测。

涌。文献[7]根据数组索引和约束变量之间的关系进行处理。然而随着数组索引的修改,工作将会繁重而复杂。文献[8]提出一种混淆算法,即在程序中插入分支代码,以明确数组变量,使得符号执行不会因为数组变量的取值无法确定而停滞。

混淆算法为处理符号执行中的数组问题开辟了新的思路,使得用符号值替换数组变量成为可能。然而该算法每次遇到数组变量时,都需要插入大量的分支,分支数的增长呈指数倍上升,大大影响了其性能和实用性。另外,该算法仅对简单的情况进行分析,较为复杂的数组问题将难以处理。

针对混淆算法的难点,本文提出一种分支混淆算法,它可减少冗余分支,并且可以处理复杂的数组问题。

2 基本概念

2.1 传统定义

符号执行中的传统概念定义如下。

定义 1(状态三元组) 符号执行状态可以用一个三元组描述为:$\langle PC, IP, S\rangle$,其中:

PC:路径条件,沿着某条路径执行积累每一步输入必须满足的约束,其形式基于输入符号的布尔型表达式。

IP:程序计数器,程序计数器定义下一个要执行的语句。

S:程序变量对应的符号值。

定义 2(一般变量) 非变数组变量称为一般变量,记为:$x\in Gel, (Gel\cap VarAry)=\emptyset$。

定义 3(使用变量) 在语句 s 中出现并且其值被使用的变量,称为使用变量,记为:$x\in Use(s)$。

使用变量在程序中可以用作条件判断、重构变量赋值、输入输出。

2.2 新增定义

为了使说明方便,条理清晰,本文特新增以下定义。

定义 4(重构变量和重构) 语句 s 中,取值有可能改变的变量,称为重构变量,记为:$x\in Rec(s)$。称 s 对 x 进行一次重构。

重构变量往往作为被赋值的对象,也可能是自增减语句中的变量。

定义 5(常数组变量) 如果数组变量的索引属于常数集合 Con,则可将该数组变量作为一般变量处理,其称为常数组变量,记为:$x\in ConAry(c), c\in Con$。

定义 6(变数组变量) 若数组变量的索引属于变量集合 Var,此时所指的内存地址无法确定,则形成了变数组变量,记为:$x\in VarAry(v), v\in Var$。

变数组变量在形式上由组内变量和组间变量构成。

定义 7(组内变量) 数组变量中用以描述索引的变量集合,用 Ind 表示。

此处所指集合,用以说明一个变数组变量的组内变量可能不止一个。此类变量关系到数组元素的内存存储地址,引发数组地址冲突,因此对组内变量的研究是工作重点。

定义 8(组间变量) 一个组间变量表示一个数组名,组间变量用 Ary 表示。

因为数组地址冲突只会发生在同名数组之中,所以研究往往在同名数组即相同的组间变量内部展开。

定义 9(一般重构) 若语句 s 中存在重构变量 i,i 属于一般变量或常数组变量,则称 s 为一般重构,记为:$s\in GelRec(i)$。

定义 10(变数组重构) 若语句 s 中存在重构变量 i,i 属于变数组变量,则称 s 为变数组重构,记为:$s\in MulRec(i)$。

定义 11(新生组内变量) 语句 s 中存在组内变量 a,a 在 s 之前被一般重构过,则称组内变量 a 为新生组内变量,以下简称新生变量。

3 混淆算法难点

符号执行数组问题研究难点主要在于,当重构变数组变量时,如何准确定位重构的内存地址,使符号值能够按照程序控制流程顺利地执行下去。

以静态数组为例,当程序声明一个长度为 n 的静态数组时:

1)如果在程序的运行过程中,该数组都以常数组的形式呈现,则可以将该静态数组视作 n 个一般变量,符号执行将无障碍地进行下去。

2)如果程序中出现了变数组变量,组间变量和组内变量就形成一类"新生变量",这使得程序在进行重构时无法确定重构的对象和内存存储位置。

仔细分析变数组变量的特点,如果组内变量集合有相等的可能性,则必然存在数组重构冲突;反之,如果可以确定每个组内变量集合各不相同,则不会出现重构冲突。混淆算法借助插入分支语句展示每种可能性,并对组内变量作出调整。然而,传统混淆算法也存在一些难点,具体表现在以下两个方面。

1)分支数剧烈增长

传统混淆算法指出,每遇到一次变数组变量,如果该变量的组内变量首次出现,则要扩展语句,插入 n 个分支,n 代表这是第 n 个首次出现的组内变量,每个分支表达不同组内变量之间的等同关系。随着数组重构语句和组内变量数的增加,各个组内变量相互之间的等同关系会呈指数增加,这无疑降低了符号执行的效率。图 1 示出多维数组变量数与插入分支动作数、插入分支数之间的对应关系。图 1 中 AryTm(Array variable time)表示变量数组出现的次数,IncAct(Times of increased branch action)是指程序中要添加的动作数,以插入 if 语句为例,在不考虑程序原本的分支条件情况下,其实就是增加 if/else 的数量。IncBra(Increased branches)是指程序中实际增添的分支路径数。从图 1 可以看出,当数组变量的数目增加 7 时,插入分支动作数和实际插入分支数剧烈增长。

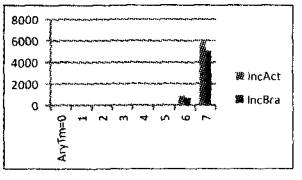

图 1 混淆算法分支变化关系

2)复杂情况下的数组问题

传统混淆算法仅讨论了一般的变数组变量重构问题,对于一般重构和变数组重构交替执行的情况并未作过多阐述,而该情况有可能产生新生变量,从而使得条件分支数急剧增长,如下所示。

```
CrossRec(int []a,int i,int j)
{
1) a[i]=1;
2) j=i+1;
```

```
3) k=i+2;
4) a[j]=2;
5) a[k]=3;
6) if(a[j]! =3)
    {
7) a[k]=4;
    }
  }
```

对于传统的混淆算法，语句 4)中的组内变量 j 与初始变量 j 等同，当算法遍历到语句 4)时，将仍然以初始变量 j 作为插入基准；而实际上语句 4)中的组内变量 j 是新生组内变量，与初始变量 j 不是同一概念，其数组的语义仍然混淆，传统的混淆算法对于该情况无法作出正确判断。

事实上，分支插入技术不必对每个变数组变量和新生变量一一进行处理。我们试图在缩减分支数与定位数组变量之间取得一种平衡，因而采取下述分支混淆算法来处理符号执行中的数组问题。

4 分支混淆算法

针对上述传统混淆算法难点，本文提出分支混淆算法，摒弃先一遍混淆、再一遍符号执行的思想，采取边混淆边符号执行的方法，即用一遍路径遍历完成分支插入和符号执行。这样一方面可以保证实时的约束求解，对于冗余分支及时剪除[9-11]，不予扩展；另一方面，符号替换可以分辨出新生组内变量，从而避免该情况下的数组语义混淆。

分支混淆算法的符号执行结构如图 2 所示。

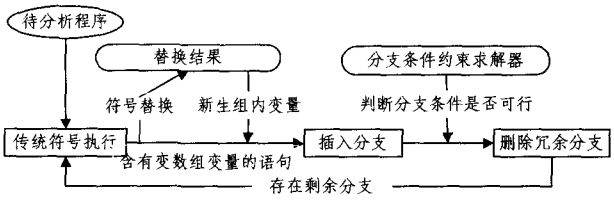

图 2 分支混淆算法的符号执行结构

分支混淆算法描述如下，算法的输入是待执行的语句序列，输出是符号执行后的语句序列。为了方便说明，该算法以数组二维变量为例。

由于不同分支上的变量数组无关存在，因此变量数组的插装仅考虑本分支上的语义混淆，仅需为每个分支维护一个数组变量集合 I，这样可以有效地减少插装的分支数。对于分支的判别，通过遍历符号树完成。

在每个分支上进行分支插装和约减的算法如下：

1)对于不存在数组二维变量的语句，采用传统的符号执行技术，对变量的符号值进行符号替换。

2)维护一个组内变量集合 I，长度为 n，I 初始化为空，n 初始化为 0。一旦遇到数组二维变量 $a[i]$，就提取组内变量 i。注意，这里的 i 可能是符号替换后的新生变量。

3)a)若 I 为空，则将 i 插入集合 I 中，不做分支扩展，完成传统符号执行。

b)若 I 不为空，且变量 i 在 I 中尚未出现，则将 i 插入集合 I 中，进入步骤 4)。

c)若 I 不为空，且变量 i 在 I 中已存在，则不做分支扩展，完成传统符号执行。

4)对于集合 I 中除当前插入的变量 i 以外的所有 n 个变量，依其插入次序，分别插装 n 个条件分支，为了便于动态约束求解，使用 if 语句。最后插入 else 语句，表示 i 与 I 中已存在的 n 个变量都不等。这样总共插入 $n+1$ 条分支语句。

对于每一个 if 语句，令其条件为 $i==I[seq]$，seq 为 n 个变量在 I 中的序列号。

5)对于步骤 4)所插入的分支条件谓词，结合符号执行之前的条件谓词，利用约束求解实时地进行判断，若该路径不可行，则不必再做以下的插入动作，再深度遍历下一个分支，重复步骤 5)。若路径可行，进入步骤 6)。

6)对于可行的条件，例如 $i==I[seq\_f]$，seq_f 为假设可行条件变量在 I 中的序号。调整重构变量 $a[i]$的组内变量，变为 $a[seq\_f]$，复制该语句以下的代码，对该范围内的代码做传统符号执行，直至下一个数组二维变量出现。

7)若路径遍历没有结束，重复步骤 3)。否则，符号执行结束。

由于程序路径在插入分支过程中动态变化，因此采用深度遍历较为合适。该算法伪代码描述如下。

```
Create(Inner); //创建组内变量集合 Inner
Initialize (Inner);// Inner 初始化为空
improved_mix ( pro )
{
  foreach sentence s in pro
  {
    if(Condition(s)==true)
    {
      Constraint(s);//对条件约束求解
      if(! Feasible(s)) s. enable==false;
      else s. enable==true;
    }
    if( Exist(x,s) &&x∈VarAry(v))
    {
      Pick (i,x); //从 x 中提取组内变量 i
      if(Rebirth(i)) //若 i 是新生变量
      {
        i=SymbolicRep(i);//符号替换 i
      }
      if(Inner == NULL)
      {
        Insert(i,Inner);
        SymbolicExec(s);//一般的符号执行
      }
      else if(Inner! =NULL && Exist(i,Inner))
      {
        SymbolicExec(s);
      }
      else
      {
        Insert(i,Inner);
        foreach element e in Inner
        {
          IfExtent(e);//if 语句扩展
        }
        ElseExtent(e);//else 语句扩展
        foreach extentedBranch eb in pro
        {
          if(eb. enable)
          improved_mix(sub_pro);//分支内部递归
```

```
        }
      }
    }
      else
      {
        SymbolicExec(s);
      }
    }
  }
}
```

5 算法讨论

分支混淆算法继承了惰性的替换思想，将复杂数组变量以一个新变量替换，摆脱了符号执行遇到复杂数组无法进行的困境。与“懒符号替换”相比，本文的改进集中在以下两个方面。

1）数组变量作为分支条件

懒替换在遇到复杂数据结构时，不做符号执行，但如果约束条件中出现复杂数据结构，约束对象的语义模糊将致使混淆问题仍然存在，甚至会得出错误的条件结果。混淆算法对于复杂数组变量的符号替换即时进行，不做延迟处理，解决了该情况下分支条件的语义模糊问题。

2）处理多索引变量数组

当一个数组的索引变量不止一个时，即在程序中出现数组名相同，但索引变量不同的情况下，懒替换会使用下标为 0 的不同变量表示这种情况。然而在不同的索引变量下，其值可能相同。懒替换无法区分这种索引的混淆。而混淆算法用增添条件约束的方法清晰地处理了这种混淆问题。

6 验证工具原型和实验结果

基于分支混淆算法，我们实现了可以处理数组问题的符号执行原型工具 ASym。ASym 采用 Parser_Generator2 工具实现前期词法分析和语法分析工作，Parser_Generator2 是 Windows 下的 lex 和 yacc 工具，可以生成 C++代码。约束求解使用 SML 求解器 Z3[12]，分支条件先被转化为 Z3 的约束模型，再调用 Z3 求解约束。本文实验环境是 Window7 OS，CPU 是 Intel Core i3-2120，主频 3.30GHz，内存 2.92GB。ASym 实现原理如图 3 所示。

ASym 首先经过代码预处理（Code Pretreatment），包括载入代码、更新代码记录等，进入编译前端模块（Compile Front-end），包括词法分析（Lexer Token）和语法分析（Parser Analysis），生成符号树类 AST_Parse 来支持分支判别函数 Branch_Def()，每判别一个新的分支就维护一个 AVS（Array Variables Set）。接着，数组变量处理模块（Array Variables Management）进行数组变量的提取（Array Variables Refine）、条件分支扩展（Condition Expansion）以及可满足性约束求解（SR，Satisfiability Reduction），SR 是通过 Z3 的 SAT 求解功能实现的。最后，符号执行过程包括对 CC（Constraint Condition）的收集，这一步是通过对数组变量的处理和代码解析共同完成的，以及对条件分支的约束求解（SML Constraint Solving）。

边插装边符号执行的意义在于可以及时判断出路径的可行性，最大限度地缩减冗余分支。应用 ASym 做了大量的关于数组程序的符号执行实验，以经典的堆排序算法为例，修复堆代码如下。

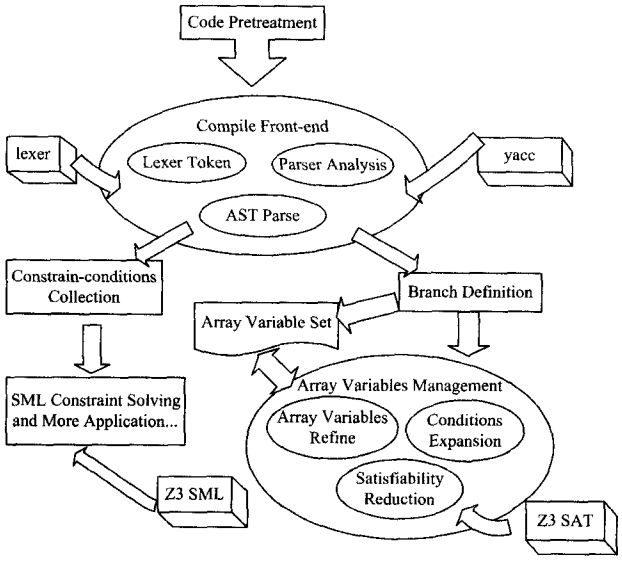

图 3 ASym 符号执行模型

```
Void FixHeap(Element L[ ],int Hsize,int root,Element k ){
1) if((2 * root)>Hsize)
2)   L[root]=k;
3) else
   {
4)  if((2 * root)==Hsize)
5)    int larger=2 * root;
6)  else if(L[2 * root]>L[2 * root+1])
7)    int larger=2 * root;
8)  else
9)    int larger=2 * root+1;
10)  if(k>L[larger])
11)    L[root]=k;
12)  else{
13)    L[root]=L[larger];
14)    FixHeap(L,Hsize,larger. k);
     }
   }
 }
```

在一次修复堆的过程中，ASym 的符号执行情况如图 4 所示。图 4 中“C”项指的是迭代的条件约束，“Cn”表示序号为 n 语句的迭代条件约束，符号“#”指增加的分支条件。限于篇幅，图 4 仅给出语句 1）－10）的执行结果。

具体结果请见 http://code.google.com/p/nkimisymbolicexecution/downloads/list。

从图 4 可以清晰地看出基于分支混淆算法的符号执行技术的特点。语句 1）－5）条件约束不予增长。语句 2）中虽然存在变数组变量，但组内变量 root 为组内变量集合中的首个变量，因此不予扩展，直接当作一般变量符号执行。对于语句 6）中出现的组内变量，需要进行分支扩展。由于该语句与语句 2）不在同一分支，因此 2 * root 为该分支首变量，仅对（2 * root+1）进行扩展。条件约束如“C”列所示，通过约束求解器的判断，符号替换只在可行条件（如语句 2）－6）中的条件）中执行下去。对于不可行的条件分支（如语句 1）－6）中的条件），符号执行将不予执行，也不再做分支扩展，用“－－”表示。而由于边执行边插装，使得语句 10）中的 large 变量可以根据不同的条件值进行准确替换。

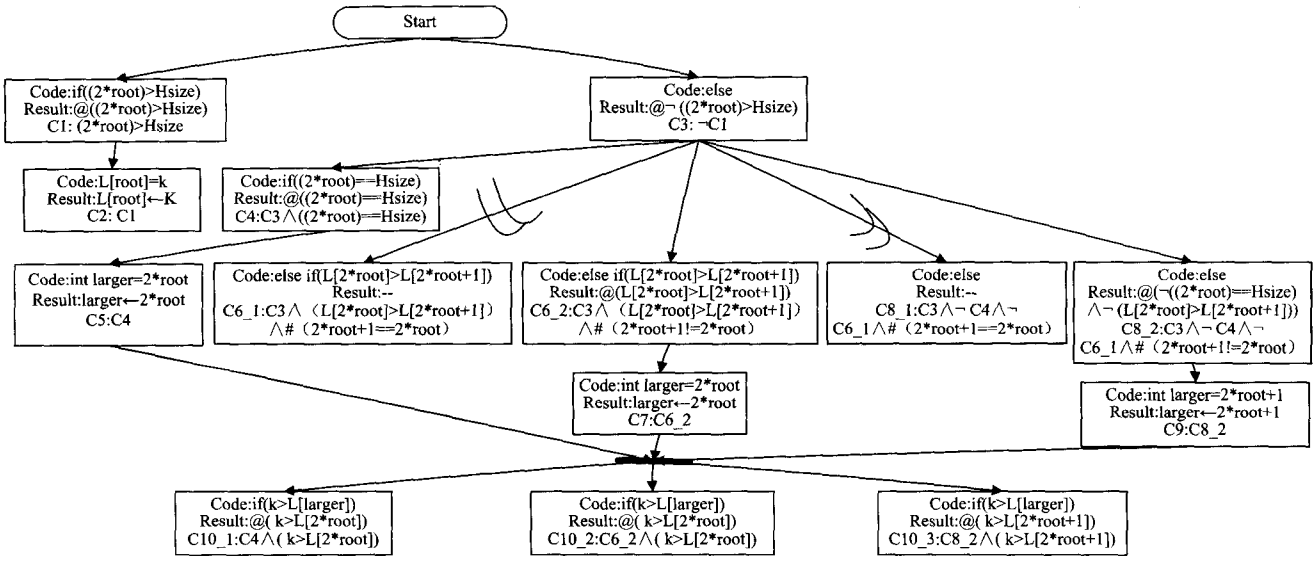

图 4　分支混淆算法的符号执行结果

表 1 为分支混淆算法性能与传统混淆算法的比较结果。表中 AlgType(Algorithm type)是算法类型，将 TCA(Traditional Confusion Algorithm)与 BCA(Branch Confusion Algrithm)进行对比。Tra(Traversal times)表示遍历次数，DealNew(Ability of deal with new array variable)是处理新生组内变量的能力。传统的混淆算法由于没有对程序原分支情况进行分析，因此 IncAct(Times of increased branch action)比较少。因其没有及时地约束求解，使得不可行分支条件(如(2 * root+1)==(2 * root))继续扩展，特别是当程序原本具有分支语句时，IncBra(Increased branches)会剧烈增长。对于不可达分支路径，分支混淆算法会插入一次分支动作，当约束求解完成之后，该路径将不再执行和扩展。因此实际的分支路径并没有剧烈增加。使用边混淆边符号执行的策略，可以及时地替换程序中的重构变量(如 large 变量)，减少执行错误。在效率方面，分支混淆仅需遍历一次待执行程序即可，传统的混淆算法需要先进行一遍混淆插入，再进行符号执行。另外，由于忽略了一般重构对于组内变量的影响，对于一般重构和变数组重构交替出现的情况，传统的混淆算法无法正确处理。而分支混淆算法可以在符号替换的基础上，不断更新组内变量的取值，使混淆算法可以正确地进行处理。

表 1　分支混淆算法与传统混淆算法性能比较

AlgType	IncAct	IncBra	Tra	DealNew
TCA	12	34	2	no
BCA	22	6	1	yes

结束语　分支混淆算法继承了传统混淆算法处理符号执行中的数组问题的优势，即可以将数组变量明确化，避免了因索引变量取值不定而造成数组变量的语义混淆。除此之外，该算法利用边混淆边符号执行和实时的约束求解策略，克服了传统混淆算法在冗余分支上的困扰，并且能够处理原程序中存在的分支结构、新生变量、数组元素作为分支条件等复杂语境，提高了符号执行的实用性。

目前，在最坏情况下(分支无法约减)，分支混淆算法的插入分支数仍是巨大的，如何进一步提高性能是下一步工作的重点。另外，ASym 仅对数组问题进行分析，对于更为复杂的数据结构如指针、结构体的研究也是未来的方向之一。

参 考 文 献

[1] King J C. Symbolic execution and program testing [J]. Commun ACM, 1976, 19(7): 385-394

[2] 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

[3] Sen K, Agha G. CUTE and Jcute: Concolic unit testing and explicit path model-checking tools[C]// Proceedings of Conference on Computer Aided Verification. 2006, 4144: 419-423

[4] Cadar C, Ganesh V, Pawlowski P, et al. EXE: Automatically generating inputs of death[J]. ACM Transaction on Information and System Security, 2008, 12: 1-38

[5] Coward P D. Symbolic execution systems-a review [J]. Software Engineering Journal, 1988, 3(6): 229-239

[6] 林梦香，陈胤立，陈睿，等. 基于懒替换的 C 符号执行[J]. 北京航空航天大学学报，2009，35(6)：687-691

[7] Zhang Jian, Wang Xiao-xu. A constraint solver and its application to path feasibility analysis[J]. International Journal of Software Engineering and Knwoledge Engineering, 2001, 11(2): 139-156

[8] 洪宇，陈光，于见平，等. 处理符号执行中数组元素混淆的一种新方法[J]. 计算机应用，2005，25(1)：434-436

[9] Ngo M N, Tan H B K. Heuristices based infeasible path detection for dynamic test data generation[J]. Information & Software Technology, 2008, 50(7/8): 641-655

[10] 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

[11] 张健. 精确的程序静态分析[J]. 计算机学报，2008，31(9)：1549-1553

[12] Z3: New High-performance Theorem Prover [EB/OL]. http://research.microsoft.com/projects/z3/

[13] 杨飚，张焕国，王后珍. 一种 C 程序内存访问缺陷自动化检测方法研究[J]. 计算机科学，2010，37(6)：155-158