支持形状分析的符号执行引擎的设计与实现

梁家彪 李兆鹏 朱 玲 沈咸飞

(中国科学技术大学计算机科学与技术学院 合肥 230026)

(中国科学技术大学先进技术研究院中国科大-国创高可信软件工程中心 合肥 230027)

摘 要 目前提高软件可靠性的方法有3种:动态测试、静态分析和程序验证。动态测试的结果依赖于测试集的设计,误报率低,漏报率高,分析结果不稳定。程序验证可以对程序的各种性质进行完备的验证。但目前程序验证通常都需要手动证明,分析成本最高。而程序静态分析可以更早、更全面、较高效和低成本地检测到程序中的缺陷。其中符号执行技术是一种比较有应用前景的静态分析技术,可以很好地控制精确度。针对符号执行可伸缩性差和容易产生路径爆炸的问题,在符号执行过程中利用形状分析技术实现自动推导循环不变式和构建函数行为规范,实现了一个较为实用的C程序分析工具。

关键词 符号执行,静态分析,循环不变式,递归函数

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.3.036

Symbolic Execution Engine with Shape Analysis

LIANG Jia-biao LI Zhao-peng ZHU Ling SHEN Xian-fei

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

(USTC-Sinovate High Confidence Software Engineering Center, Institute of Advanced Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract Dynamic testing, static analysis and formal verification are three major methods for improving the dependability of software. Since the result of dynamic testing depends on the design of test case suite, it is unstable and can lead to high false negative rate. Formal verification is a complementary method for proving the correctness of software. So far most of the proof still need to be implemented by hand which makes formal verification hard with high cost. Static analysis is an efficient and low-cost method for detecting bugs in software. One of the promising techniques of static analysis is symbolic execution, while it has a good control over the degree of accuracy. Targeting at the path explosion problem and poor scalability of symbolic execution, and taking advantage of shape analysis, loop invariants and function specification inference during symbolic execution, we implemented an efficient analysis tool for C programs.

Keywords Symbolic execution, Static analysis, Loop invariant, Recursive function

1 引言

快速有效地提高和确保软件的安全质量和可靠性一直都是软件开发过程中的一个迫切需求。提高软件安全质量和可靠性的常见方法有3类:程序验证、动态测试和静态分析。程序验证通过形式化方法,如Hoare逻辑[1]和分离逻辑[2]等,对程序的各种性质进行严格的证明,是保证程序正确性的最严格的手段。程序验证的证明过程目前仍无法实现完全的自动化,大部分需要人员手动证明,这使得程序验证的投入成本极高且周期过长,目前还无法在工业界中大规模应用。动态测试和静态分析已经在实践中有很好的应用。动态测试通过构造测试集作为程序的输入,运行程序来验证程序软件的动态行为和结果的正确性。动态测试结果的好坏依赖于测试集的设计。缺乏良好的测试集,动态测试容易出现程序覆盖率低和漏报率高的情况,且不同的程序需要不同的测试集,这会使得分析结果不稳定。而程序静态分析不需要运行程序,通过分析源代码,自动收集并分析相关信息来检测程序中的各种缺陷。相比之下,静态分析可以更早、更全面、更高效和更低成本地检测到程序中的缺陷。

比较成熟的静态分析技术有:模型检查[3]、数据流分析[4]、抽象解释[5]和符号执行[6]。其中符号执行被广泛用于为待测试程序自动生成测试用例,取得了很好的效果。符号执行在分析过程中可对特定部分的程序变量的取值进行符号化。符号化的变量的取值范围被路径约束限制,只要一条执行路径的路径约束不使符号化变量的取值出现矛盾或触发非预期的程序行为,符号执行就会尝试分析该执行路径。符号执行可做到较全面和较精确的分析,目前基于符号执行的比较优秀的程序分析工具有KLEE[7]、Clang Static Analyzer[8]

到稿日期:2015-02-06 返修日期:2015-05-18 本文受国家自然科学基金面上项目(61170018)资助。

梁家彪(1987—),男,硕士生,主要研究方向为软件安全;李兆鹏(1978—),男,副研究员,主要研究方向为程序语言、程序验证、出具证明编译器和自动定理证明;朱 玲(1989—),女,硕士生,主要研究方向为软件安全;沈咸飞(1990—),男,硕士生,主要研究方向为软件安全。

和 Saturn[9]。从实践来看，符号执行和其它静态分析技术一样，也面临着如下问题：

• 误报率和漏报率。为提高分析速度，静态分析技术通常需要对程序状态进行一定程度的抽象，而该操作会弱化程序的约束条件，造成分析不精确，产生误报和漏报。

• 路径爆炸。符号执行会尽可能地尝试分析程序中的各个分支路径，而程序中分支路径的数目通常会随着程序规模的增长而呈指数级增长。路径爆炸会使得符号执行的分析性能显著降低。如何减缓路径爆炸及优化路径选择策略是符号执行需要考虑的一个重点。

• 可伸缩性。路径数目的暴增及频繁的过程间调用很容易使得符号执行的分析效率陡降。良好的可伸缩性是决定符号执行能否应用于大规模程序的主要因素。

本文针对上述问题，在符号执行过程中增加新的分析技术手段，实现一个分析效率更高的针对 LLVM IR[10] 代码的符号执行引擎。本文的主要工作如下：

(1)实现函数行为规范的自动构建。函数行为规范使用形式化语言描述函数产生的行为效果，利用函数行为规范可实现快速复用一个函数已有的分析结果，提高分析过程的可伸缩性。

(2)在符号执行过程中使用了形状分析技术，可以对程序状态进行较为抽象的描述，但又不至于丢失过多的信息。

(3)优化循环体、递归函数的分析。在分析过程中，本文自动尝试推导循环不变式和递归函数的行为规范。相比于利用常见的简单展开循环体、递归函数有限次数进行分析，利用循环不变式、递归函数的行为规范能更全面地覆盖程序的各种状态，且可以减缓路径爆炸问题，提高分析效率。

(4)实现了一个分析 LLVM IR 字节码的符号执行引擎 ShapeChecker，并对 GNU Coreutils[11] 进行了测试，取得了一些初步的成果。

本文第 2 节对符号执行引擎 ShapeChecker 的框架进行介绍；第 3 节介绍在分析过程中为支持形状分析所使用的程序状态描述；第 4 节介绍在过程间分析中使用的函数行为规范；第 5 节介绍对循环体、递归函数的处理，主要为自动推导循环不变式和递归函数的行为规范的实现；第 6 节是实验结果；第 7 节是相关工作的比较；最后是总结和后续工作展望。

2 工具的框架

本文符号执行引擎的整体框架如图 1 所示。目前的工具主要针对 C 程序进行分析，C 源代码经过 Clang 编译后转变为 LLVM IR 代码，符号执行引擎通过解析 LLVM IR 代码进行分析。经过 Clang 编译的代码可以预先排除掉许多明显的错误(如语法错误、类型错误)。而本文的分析工具专注于检测一些不易发现的程序缺陷，如过程间调用引起的内存越界访问、空指针引用、内存泄漏和形状错误等。

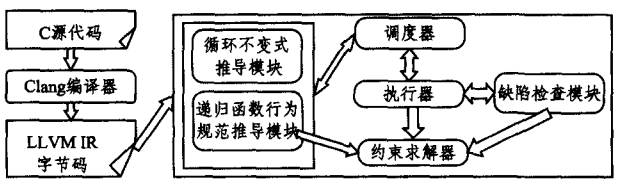

图 1 ShapeChecker 的框架

图 1 中的执行器是整个符号执行引擎的核心模块，其它模块在执行器的调度下协同完成整个分析过程。执行器的核心内容是实现一个如图 2 所示的算法(简化版)。在图 2 中一个程序状态被简略地描述为(l, pc, s)，l 指代程序的当前指令，pc (Path Constraint)指代程序当前满足的路径约束，s 指代程序当前的内存空间。执行器每次接受一个程序状态输入，并根据该程序状态的当前指令采取相应的动作。在图 2 中，第 5 行表示一个变量的值被更新了；第 7 行表示分支跳转；第 14 行表示约束检查，检测状态能否满足约束条件 e。

```
1. W := (l0, true, s0)
2. while W≠∅ do
3.   (l, pc, s) = pickNext(W); Sd = ∅; Sd = ∅
4.   switch (instr(l))
5.   case v := e;
6.     Sa := {(succ(l), pc, s[e ↦ eval(s,e)])}
7.   case br e lt lf;
8.     if pc∧e may be true then
9.       Sa := {(lt, pc∧e, s)}
10.    end if
11.    if pc∧¬e may be true then
12.      Sa := Sa∪{(lf, pc∧¬e, s)}
13.    end if
14.  case assert(e);
15.    if pc∧e is not statifiable then
16.      Sd := {(l, pc, s)}
17.    else
18.      Sa := {(l, pc, s)}
19.    end if
20.  end switch
21.  W := W∪Sa/Sd
22. end while
```

图 2 符号执行引擎的主体算法

调度器模块的主要作用是维护一个图 2 中所示的可执行状态集合 W。调度器模块提供了不同的调度策略，如并行调度。同时调度器模块还配合循环不变式、递归函数行为规范推导模块实现循环不变式和递归函数行为规范的推导。

约束求解器模块提供约束问题求解的功能，目前约束求解器是基于 STP[12] 实现的，并在 STP 的基础上初步实现对形状谓词的支持。

3 程序状态与断言语言

本节介绍为支持形状分析而在符号执行引擎中使用的程序状态表示。

3.1 断言语言

本文使用了新的支持形状描述的断言语言，如图 3 所示。

Assertion	assert::=	true|false|term1 rop term2|pred
Relation Op	rop::=	==|!=|<|<=|>|>=
Predicate	pred::=	lseg(term1, term2, tset)|
		dlseg(term1, term2, term3, term4, tset)|
		tree(term, tset)|valid(tset)|
		separated(tset(, tset)*)|fresh(tset)|
		freshbyte(term1, term2)|allocable(tset)|
		freeable(tset)

图 3 支持形状描述的断言语言

term 表示变量取值或由算术运算符构成的表达式；tset 是内存地址集合，其中的元素也是 term；pred 是用于描述内存空间的抽象谓词；valid 表示指定的 tset 中元素都对应了一个有效的内存对象；separated 指明多个 tset 间不存在交集；fresh 和 freshbyte 用于表示相应的内存空间是新动态分配的；allocable 表示指定的 tset 中的各个元素对应的内存空间都已被手动释放；freeable 表示指定的 tset 中的各个元素对应的内存空间都是可手动释放的；lseg、dlseg 和 tree 都是用于描述内存空间的数据结构形状的谓词，分别用于抽象描述单链表、双链表和二叉树数据结构。通过抽象谓词可以对特定部分的内存空间形成抽象描述，从而有助于推导循环不变式和递归函数的行为规范。

为方便后文的讨论，式(1)给出了 lseg 的递归定义，关于断言语言更详细的介绍请参考文献[13]。式(1)中 p 是一个单链表表段的首节点，q 是表段的末尾节点的后继，s 是该表段所含的节点的集合；$next(p)$表示单链表节点中指向后继节点的指针域。

$$lseg(p,q,s)\overset{def}{=}(p==q\wedge s==\emptyset)\vee(valid(\{p\})\wedge next(p)==t'\wedge lseg(t',q,s')\wedge s==\{p\}\cup s'\wedge p\notin s') \quad (1)$$

3.2 程序状态

在符号执行过程中，需要记录的关键信息有：

• 下一条将执行的指令，记为 l。

• 当前执行路径满足的约束条件，记为 pc，主要是程序中的 if-else 分支条件构成的对变量的取值的约束。

• 程序当前点的内存空间状态，即记录式(2)和式(3)中的映射关系，记该部分信息为 M。

$$Variable\rightarrow Address/Value \quad (2)$$

$$Address\rightarrow Address/Value \quad (3)$$

每个程序变量、动态分配的内存空间都会被视为不同的独立的内存对象。程序状态的内存空间保存了这些内存对象的首地址、大小及各个偏移位置的取值等信息。其中内存对象的首地址和大小都是符号化的，并不需要分配真实的物理内存空间。

为了支持形状分析，本文的符号执行引擎将内存空间 M 划分为两部分：具体的(Concrete)内存空间 M_c 和抽象的(Abstract)内存空间 M_a。M_c 记录式(2)和式(3)中的映射关系。M_a 是使用图 3 中的抽象谓词对特定内存空间形成的抽象表示。在特定的程序点，数据结构在形状上的特质是关注的重点，通过发现数据结构的形状特性，有助于推导循环不变式和递归函数的行为规范。当 M_c 中的某部分内存空间可满足特定的形状的性质时，如这部分内存空间在形状上构成了一个单链表或二叉树等，这部分内存空间就会从 M_c 中剥离。剥离出来的内存将会使用形状谓词进行抽象表示，并记录在 M_a 中。而当某条程序指令需要访问被抽象到 M_a 中的内存空间时，对应的抽象内存空间就会按需展开到 M_c 中。

具体内存空间能够保存足够精确的信息，在分析过程中会尽量避免将具体内存空间中的内容抽象到抽象内存空间中；而在分析循环体和递归函数时，可以利用抽象内存空间发现不变的性质。具体内存空间和抽象内存空间的划分和配合可以使分析过程在精度和效率上取得很好的平衡。

4 函数行为规范

对于过程间分析，利用函数行为规范使得过程间分析能复用已有的分析结果，提高分析过程的可伸缩性。本节对函数行为规范的表示及其自动构建的实现进行介绍。

在处理被调用函数时，常用的方法是展开被调用函数执行。但是该方法每次都需要重新展开被调用函数执行且其分析结果无法很好地利用于后续分析中，可伸缩性差。而本文在分析过程中为每个函数自动构建行为规范，使一次分析结果可以在多处快速复用。相比展开函数执行，利用函数行为规范在运行时间和可伸缩性上更具优势。每个函数行为规范包含的内容如下：

• 前条件。根据调用点的上下文为被调用函数的参数构造约束条件。该部分约束条件主要是对函数的指针参数进行概要描述，如指针参数是否有效及其指向的内存空间的大小等。

• 守卫条件。随着对函数分析的不断深入，函数的各个可执行路径对函数参数的约束条件不断增多，这部分新增的对函数参数的约束条件称为守卫条件。

• 后条件。用于描述该函数被调用后会对调用者产生的影响。如函数返回值，对指针参数指向的内存的修改等。

前条件是一个函数的起始约束，约束条件比较弱，用于预先过滤一部分不需要进行分析的情况。在一个前条件约束下，通常存在多条可执行路径，每条可执行路径都会产生不同的守卫条件，增强对函数行为的描述。每条可执行路径都对应一个函数行为规范。

在分析过程中使用的形状分析技术增强了函数行为规范的表达能力。如图 4 所示的代码，借助于形状谓词 lseg，可以自动构造出一个如表 1 所列的函数行为规范。

```
1. typedef struct List{int v;struct List * next;}List;
2. List * createList(unsigned x){
3.   List * p,* t;int i;
4.   if(x==0){return NULL;}
5.   p=(List*)malloc(sizeof(List));
6.   t=p;t->next=Null;i=1;
7.   while(i<x){
8.     t->next=(List*)malloc(sizeof(List));
9.     T=t->next;t->next=NULL;i+=1;
10. }
11. return p;
12. }
```

图 4 代码示例：创建单链表

表 1 createList()的一个行为规范

前条件	true
守卫条件	x>0
后条件	lseg(res,NULL,s)∧fresh(s)

表 1 的行为规范说明了函数 createList()的返回值(用 res 指代)指向了一个单链表，且该单链表的尾节点的 next 域为 NULL；fresh(s)强调该单链表不为空，且节点都是动态分配的。该行为规范符号化了返回链表的长度，可以覆盖所有满足 x>0 的情况。该行为规范简单，在调用 createList()时利用该行为规范要比展开执行 createList()快。在实际测试中，该行为规范也保留了较好的精确度；在检测内存泄漏、空

指针引用的缺陷方面，利用形状分析的分析结果要明显好于未使用形状分析的分析结果。

5 循环不变式及递归函数行为规范的推导

对于循环体、递归函数的分析，常见的方法是对其进行有限次数的展开，如 KLEE 和 Clang Static Analyzer 的做法。有限次数展开循环体、递归函数难以实现全面的状态覆盖。本文在分析循环体、递归函数时采用了自动推导不变式、行为规范的方法，以对程序进行更全面的考虑，同时减缓状态爆炸问题，提高分析效率 。

5.1 形状谓词折叠

在推导不变式的过程中，一个重要的步骤是对程序状态进行抽象，再在抽象后的程序状态上发现循环、递归函数的数据结构在形状上不变的特性。图 5 给出了 lseg 的抽象规则。规则 SingleNodeLseg 将具体内存空间中的一个单链表的节点抽象为抽象内存空间中的一个表段；规则 MergeTwoLseg1和 MergeTwoLseg2 将两个表段合并为一个表段；规则 SimplifyLseg1 和 SimplifyLseg2 将一个表段上的指针集合抽象为一个符号集合。

(SingleNodeLseg)

$$\Gamma \vdash p:typ*\quad typ=type\{\overline{typ_i^{\,ri}}\}\ PtrSet(typ)=\{next\}$$
$$\frac{P\Leftrightarrow[getelementptr\ p\ idx]==data\ (0\leqslant idx<i, idx\neq next)\wedge P'}{P\wedge[getelementptr\ p\ next]==q \rightsquigarrow P'[q/[getelementptr\ p\ next]]\wedge lseg(p,q,\{p\})}$$

(MergeTwoLseg1)

$$\frac{P\Rightarrow\{p\}\cap\{q\}==empty}{P\wedge lseg(p,q,\{p\})\wedge lseg(q,r,\{q\})\rightsquigarrow P\wedge lseg(p,r,\{p,q\})}$$

(MergeTwoLseg2)

$$\frac{P\Rightarrow s\cap\{q\}==empty}{P\wedge lseg(p,q,s)\&\&lseg(q,r,\{q\})\rightsquigarrow P\wedge lseg(p,r,\{p\}\cup s)}$$

(SimplifyLseg1)

$$\frac{s\ is\ fresh\ P\Leftrightarrow fresh(p)\ \&\&\ fresh(q)\wedge P'}{P\wedge lseg(p,r,\{p,q\})\rightsquigarrow P'\wedge lseg(p,r,s)\wedge fresh(s)}$$

(SimplifyLseg2)

$$\frac{s'\ is\ fresh\ P\Leftrightarrow fresh(s)\wedge fresh(q)\wedge P'}{P\wedge lseg(p,r,s\cup\{q\})\rightsquigarrow P'\wedge lseg(p,r,s')\wedge fresh(s')}$$

图 5 lseg 的部分抽象规则

5.2 循环不变式的推导

设一个循环的循环条件为 b，循环体为 c。循环的执行过程如式(4)所示。

$$\{S_{pre}\}\ \textbf{while}\ b\ \textbf{do}\ c\{S_{post}\} \tag{4}$$

其中，S_{pre} 和 S_{post} 分别表示执行循环前和执行循环后的程序状态。循环体的不变式的自动推导算法的主要思想如下：

1)令 $S_0'=S_{pre}$，$i=0$。

2)根据已分析得到的 S_0'，…，S_i'，计算得到 S_{i+1}，使得 $\{S_i'\wedge b\}c\{S_{i+1}\}$。

3)对 S_{i+1} 进行抽象，得到 S'_{i+1}，使得 $S_{i+1}\Rightarrow S'_{i+1}$。

4)若 $S'_{i+1}\Rightarrow U_iS_i'$，则终止迭代过程；否则令 $i=i+1$，返回步骤 2)继续迭代。

若步骤 4)中 $S'_{i+1}\Rightarrow U_iS_i'$ 成立，则使用 $U_iS_i'\wedge\neg b$ 作为执行该循环体后得到的程序状态。

步骤 3)是推导不变式的关键，循环体的抽象动作包括检测循环体是否操作了像单链表这样可抽象描述的数据结构。若有，则每次迭代次数结束后，将这部分数据结构从具体内存空间抽象到抽象空间。最后应用谓词抽象规则，对抽象内存空间中的抽象谓词进行进一步抽象，如图 5 中的 SimplifyLseg 规则。

以图 4 中 createList()函数的循环体为例，说明循环不变式的推导。

在进入循环体前，利用规则 SingleNodeLseg，程序状态的抽象表示为 S_0：$lseg(p,\text{NULL},\{p\})\wedge i=1$。

第一次迭代后，利用规则 SingleNodeLseg、MergTwoLseg1和 SimplifyLseg1，程序状态的抽象表示为 S_1：$lseg(p,t,\{p\})\wedge lseg(t,\text{NULL},\{t\})\wedge i=2$。

第二次迭代后，利用规则 SingleNodeLseg、MergeTwoLseg2和 SimplifyLseg2，程序状态的抽象表示为 S_2：$lseg(p,t,s_1')\wedge fresh(s_1')\wedge lseg(t,\text{NULL},\{t\})\wedge i=k_1'\wedge k_1'<x$。其中将 i 的值和 p 指向的表段的地址集合都抽象为了符号值。

继续进行第三次迭代，利用与步骤 3)中一样的抽象规则，程序的抽象表示为 S_3：$lseg(p,t,s_2')\wedge fresh(s_2')\wedge lseg(t,\text{NULL},\{t\})\wedge i=k_2'\wedge k_2'<x$，则迭代过程已经到达不动点，终止迭代过程，循环不变式为 $S_0\vee S_1\vee S_2$。

在实际的处理过程中，本文还考虑了循环体中存在由 C 语言中的 break、return 语句造成循环体中断的情况。在迭代过程中会记录循环体每次迭代在 break、return 语句形成的出口处的状态，当循环体迭代到不动点时，我们认为在这些出口点的状态也达到了不动点。这些状态也作为 S_{post} 中的一部分内容继续往后执行。

5.3 递归函数的不变的推导

本文在处理递归函数时也进行了递归函数的行为规范的自动推导。与推导循环不变式的原理相同，通过迭代执行递归函数，侧重发现操作的数据结构在形状上的不变特性，从而构造出递归函数的行为规范。在对递归函数进行迭代分析时，需要将递归函数 RF 的执行路径区分为两类：包含递归调用的可执行路径 P_{rec} 和不包含递归调用的可执行路径 P_{nrec}。递归函数的不变式的推导算法如下：

1)设 S_{pre} 为递归函数 RF 在调用点的初始状态。计算 S_0，使得$\{S_{pre}\}P_{nrec}\{S_0\}$。令 $i=0$。

2)计算 S'_{i+1}，使得$\{S_{pre}\}P_{rec}\{S'_{i+1}\}$。在计算过程中，使用$\{S_{pre}\}RF\{S_i\}$作为递归函数 RF 的行为规范。

3)对 S'_{i+1}进行抽象得到 S_{i+1}，满足 $S'_{i+1}\Rightarrow S_{i+1}$。

4)若 $S_{i+1}\Rightarrow U_iS_i$，则终止迭代过程，使用$\{S_{pre}\}RF\{U_iS_i\}$作为递归函数 RF 的最终行为规范；否则，令 $i=i+1$，转步骤 2)。

通过该算法，可以主动尝试为递归函数构建行为规范，在后续分析过程中可以减少重复的展开执行。

6 实验结果

本文以 KLEE 为基础，实现了一个分析 C 程序的符号执行引擎 ShapeChecker。该分析工具在分析过程中利用形状分析技术自动推导循环不变式和构建函数(包括递归函数)的行为规范，并取得了较好的分析结果。

符号执行过程中，状态爆炸问题很容易引起内存消耗的急剧增加，这样的问题在 KLEE 中很容易遇到。相比之下，ShapeChecker 的内存消耗要比 KLEE 少很多。本文随机选取了 GNU Coreutils 中的部分程序进行了测试，比较了

KLEE 和 ShapeChecker 的内存消耗，结果如图 6 所示。

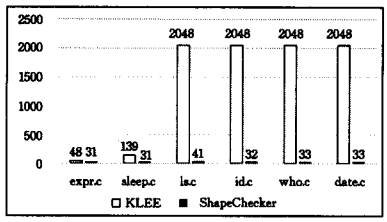

图 6　KLEE 和 ShapChecker 的内存消耗比较

在测试过程中，将内存上限设置为 2GB，图 6 的纵坐标表示最大内存消耗，单位是 MB。测试结果中，除了 expr. c 和 sleep. c，KLEE 的内存消耗都超出了上限，而 ShapeChecker 的内存消耗平均为 33MB。出现这样的现象是因为：

1）如第 3 节中提到的，ShapeChecker 支持对动态分配的内存对象的首地址和大小进行符号化，在分析过程中无需为动态内存空间分配请求分配真实的物理空间，而 KLEE 在处理动态内存空间分配请求时是需要真实分配物理空间的。

2）利用函数行为规范，ShapeChecker 可以避免重复对函数进行展开分析，利用函数行为规范所需的内存空间要比直接展开被调用函数执行所需要的内存空间少。

3）循环不变式和递归函数的行为规范的使用可以减缓状态爆炸问题，减少分析的状态的数目，下文会有讨论。

循环体、递归函数的不变式的利用有助于减缓状态爆炸问题。如图 7 中的代码（结构体 struct List 和函数 createList（）见图 4），main（）中的变量 len 的取值是未知的。因为 KLEE 在分析循环体时需要知道确定的循环次数，所以在执行 main()中的第 14 行代码的被调用函数时需要对 len 的所有可能取值逐一考虑，会产生 255 个分支状态。而ShapeChecker 在推导循环不变式时不用关心 len 的具体取值，所以在执行 main()中的第 14 行代码时不用形成新的分支状态。ShapeChecker 执行图 7 中的代码最终只产生 3 个分支状态，要比 KLEE 产生的状态少很多。

```
1. void deleteList(List * l){
2.   List * tmp;
3.   if(l==NULL){return;}
4.   tmp=l->next;
5.   while(tmp!=NULL){
6.     free(l);l=tmp;tmp=tmp->next;
7.   }
8.   free(l);
9. }
10. int main(){
11.  List * list;int len;
12.  scanf("%d",&len);
13.  if(0<len&&len<256){
14.      list=createList(len);deleteList(list);
15.  }
16.  return 0;
17. }
```

图 7　创建和删除单链表

形状分析技术有助于提高分析结果的准确性，如图 7 中的程序。在注释掉第 8 行代码后，单链表的最后一个元素未能释放，产生内存泄漏。Clang Static Analyzer 和 Klee 都未能检测出该问题。而 ShapeChecker 通过形状分析和循环不变式可以检测到 deleteList()中发生了内存泄漏。

当函数体的控制流图较为复杂时，如含有循环体或者该函数是个递归函数，利用函数行为规范要比直接展开函数执行快。考虑赋给 len 一个固定的取值，重复执行图 7 中 main（）的第 14 行代码，图 8 给出了不同执行次数下 KLEE 和 ShapeChecker 所需的时间。图 8 中的横坐标表示执行 main（）中的第 14 行代码的次数；纵坐标表示执行时间，单位为 ms。ShapeChecker 因为在第一次执行 createList（）和 deleteList()时需要推导循环不变式，所以要比 KLEE 慢。但在后续分析时，ShapChecker 会直接利用 createList（）和 deleteList()已有的行为规范，而 KLEE 仍会展开被调用函数，在执行速度上 KLEE 要慢于 ShapChecker。

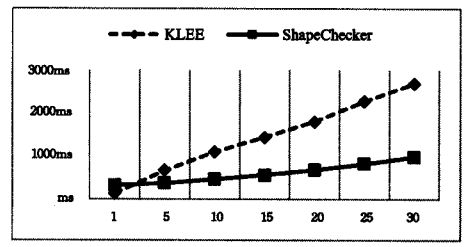

图 8　KLEE 和 ShapeChecker 的执行时间比较

本文工具会在如下情况产生误报：实现代码的错误、定理证明器超时或错误、不支持或者描述不够精确的库函数规范、不支持的 C 特性等。为了说明工具的误报率情况，选取了 Verisec[14] 部分评测程序对 ShapeChecker、KLEE 和 Clang Static Analyzer 进行了评测，表 2 给出了测试结果。其中 Verisec 是一个 C 程序静态分析工具的测试套件，该套件中的例程一般来源于发现的开源软件中的真实漏洞，可用于评估静态分析工具在检测缓冲区溢出错误时的精确度。表 2 中 Y 表示工具能正确检测出错误和没有产生误报，N 表示工具未能检测出程序中的错误，F 是对没有错误的程序产生了误报。在误报率方面，KLEE 需要更多用户输入辅助分析，因而误报率比本文工具要低。与 Clang Static Analyzer 相比，本文工具因采用了路径敏感、内存描述和形状谓词等精确分析技术而误报率较低。此外，经过分析，ShaperChecker 产生误报的原因是对字符串描述不够精确，该问题已列为工具后续改进的一个方向。

表 2　检测缓冲区溢出错误

测试程序	Shape Checker	KLEE	Clang Static Analyzer
Apache/full_bad. c	Y	Y	N
Apache/full_ok. c	Y	Y	Y
Apache/full_ptr_bad. c	Y	Y	N
Apache/full_ptr_ok. c	Y	Y	Y
Apache/simpl_bad. c	Y	N	N
Apache/simpl_ok. c	Y	Y	Y
Apache/simp2_bad. c	Y	N	N
Apache/simp2_ok. c	Y	Y	Y
Apache/simp3_bad. c	Y	N	N
Apache/simp3_ok. c	Y	Y	Y
Apache/strncmp_bad. c	Y	N	N
Apache/strncmp_ok. c	Y	Y	Y
Apache/iter1_prefixLong_arr_bad. c	Y	Y	N
OpenSER/guard_strchr_bad. c	Y	Y	N
OpenSER/guard_strchr_ok. c	F	Y	Y
OpenSER/guard_strstr_bad. c	Y	Y	N
OpenSER/guard_strstr_ok. c	F	Y	Y
OpenSER/guard_random_index_bad. c	Y	Y	N
OpenSER/guard_random_index_ok. c	F	Y	Y

7 相关工作

目前利用符号执行技术的工具已有不少，本节将对其中的几个工具与 ShapeChecker 进行比较讨论。

KLEE 在分析过程中需要为每个内存对象分配真实的内存空间，当调用外部函数时就利用 LLVM 的执行引擎 JIT 在这些真实的内存空间上执行被调用函数。KLEE 不支持符号化大小的内存空间。Clang Static Analyzer 使用了文献[15]中提到的内存模型，同样不支持符号化大小的内存对象。ShapeChecker 支持符号化大小的内存对象，使得分析过程考虑的情况会更全面。

KLEE 在分析循环体时会一直循环执行，直到循环条件不成立。当循环条件中含有符号变量时，KLEE 就会尝试所有可能的情况，这使得路径爆炸问题在 KLEE 中很严重；而 Clang Static Analyzer 则是简单展开循环体几次进行分析，分析的情况不全面。ShapeChecker 通过利用循环体不变式可以有效减缓路径爆炸问题，同时实现更全面的程序状态覆盖。

Saturn 和 SMART[16] 在分析过程中都利用了类似函数行为规范的技术，并在提高分析过程的可伸缩性上取得了较好的结果。但 Saturn 在分析过程并未利用形状分析技术，所以在分析单链表这样的动态数据结构时会产生很高的误报率（>95%）[17]。SMART 根据函数的调用关系，使用自底向上的方式进行分析，即若函数 f 调用了函数 g，则先分析并构造 g 的行为规范，再利用 g 的行为规范分析 f。该分析顺序需要被分析函数考虑所有可能的输入情况，包括许多程序中不会出现的情况。而 ShapeChecker 采用了延迟构建函数行为规范的方式，即当执行到 g 在 f 中的调用点时才分析 g。在该方式下，可以根据 g 在 f 中的调用点的上下文为 g 构建出一个前条件，通过该前条件可以过滤掉一部分不需要分析的情况。

KLEE 在分析过程中通常手动修改源代码，指明哪些变量的取值是符号化的；Saturn 处理循环体时需要手动提供循环不变式。而 ShapeChecker 在分析过程中同 Clang Static Analyzer 一样，无需用户提供额外的辅助信息（例如手动提供循环不变式、指明数据类型的形状等），提高了工具的易用性。

结束语 路径爆炸问题仍是本符号执行引擎面临的重要难题，对到达程序中同一位置的不同状态进行合并是一种有效的方法。本文在推导循环不变式和构建函数行为规范的过程中发现不少程序状态是十分相近并能合并为同一个状态的。在后续的工作中判断两个状态是否适合进行合并及如何实现合并将是本符号执行引擎的研究重点，这将会给整个分析过程带来很大的优化。目前本符号执行引擎还只能处理单链表、双链表和二叉树这些简单的数据结构。支持更复杂的数据结构，如嵌套链表，也是本符号执行引擎考虑的重点。

本文讨论了利用形状分析技术实现自动推导循环体不变式和函数（含递归函数）行为规范的可行性，并以 KLEE 为基础实现了一个较为实用的分析工具 ShapeChecker。形状分析技术能实现对程序状态进行较为精确的抽象描述，降低误报率和漏报率；循环不变式和函数行为规范的构建能避免大量的重复分析，提高分析效率。但目前 ShapeChecker 仍是一个比较初步的工具，如支持的抽象的数据结构还比较少，约束求解器的求解能力比较弱而无法支持更复杂的断言语言等，这都是我们后续研究的方向。

参 考 文 献

[1] Hoare C A R. An axiomatic basis for computer programming [J]. Communications of the ACM, 1969, 12(10): 576-580

[2] Reynolds J C. Separation logic: A logic for shared mutable data structures[C]// 17th Annual IEEE Symposium on Logic in Computer Science, 2002. IEEE, 2002: 55-74

[3] Clarke E M, Emerson E A. Design and synthesis of synchronization skeletons using branching time temporal logic[M]. Springer Berlin Heidelberg, 1982

[4] Kildall G A. A unified approach to global program optimization [C]// Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, 1973: 194-206

[5] Cousot P, Cousot R. Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]// Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, 1977: 238-252

[6] King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394

[7] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]// OSDI. 2008, 8: 209-224

[8] Clang Static Analyzer[OL]. http://clang-analyzer.llvm.org/

[9] Dillig I, Dillig T, Aiken A. Sound, complete and scalable path-sensitive analysis[J]. ACM SIGPLAN Notices, ACM, 2008, 43(6): 270-280

[10] Llvm I R. LLVM Language Reference Manual[OL]. http://llvm.org/docs/LangRef.html

[11] Coreutils. GNU core utilities[OL]. http://www.gnu.org/software/coreutils

[12] Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays[M]// Computer Aided Verification. Springer Berlin Heidelberg, 2007: 519-531

[13] ShapChecker [OL]. 2015. http://kyhcs.ustcsz.edu.cn/~shapechecker

[14] Ku K, Hart T E, Chechik M, et al. A buffer overflow benchmark for software model checkers[C]// Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. ACM, 2007: 389-392

[15] Xu Z, Kremenek T, Zhang J. A memory model for static analysis of C programs[M]// Leveraging Applications of Formal Methods, Verification, and Validation. Springer Berlin Heidelberg, 2010: 535-548

[16] Godefroid P. Compositional dynamic test generation[C]// Proceedings of POPL. 2007

[17] Dillig T. A Modular and Symbolic Approach to Static Program Analysis[D]. Palo Alto: Stanford University, 2011