基于输入约束的符号执行优化

汪孙律，林渝淇，杨秋松，李明树
（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）

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，提出了基于输入约束的符号执行（ICBSE）优化框架。该方法通过分析程序代码自动提取 3 类输入约束，随后使用这些约束引导符号执行更关注于核心功能代码。在 KLEE 中实现了上述优化框架，并对 coreutils、binutils、grep、patch、diff 这 5 个程序套件中的 7 个常用程序做了检测。ICBSE 发现了 7 个之前未知的缺陷（KLEE 只检测其中 3 个）。同时，ICBSE 将指令行覆盖率、分支覆盖率分别提升了约 20%，时间开销降低了约 15%。

关键词：符号执行；输入约束；路径爆炸；缺陷查找
中图分类号：TP311
文献标识码：A
doi: 10.11959/j.issn.1000−436x.2019062

Symbolic execution optimization method based on input constraint

WANG Sunlyu, LIN Yuqi, YANG Qiusong, LI Mingshu
National Engineering Research Center of Fundamental Software, Institute of Software Chinese Academy of Sciences, Beijing 100190, China

Abstract: To solve path explosion, low rate of new path's finding in the software testing, a new vulnerability discovering architecture based on input constraint symbolic execution (ICBSE) was proposed. ICBSE analyzed program source code to extract three types of constraints automatically. ICBSE then used these input constraints to guide symbolic execution to focus on core functions. Through implemented this architecture in KLEE, and evaluated it on seven programs from five GNU software suites, such as coreutils, binutils, grep, patch and diff. ICBSE detected seven previously unknown bugs (KLEE found three of the seven). In addition, ICBSE increases instruction line coverage/branch coverage by about 20%, and decreases time for finding bugs by about 15%.

Key words: symbolic execution, input constraint, path explosion, bug finding

1 引言

符号执行是一种经典程序分析技术，它将程序中的变量抽象为符号，通过符号的形式模拟程序运行，收集程序路径上的约束条件[1-2]。使用该技术可以遍历程序路径空间，检查程序是否满足某些安全特性。符号执行当前多用于辅助软件测试[3]、程序缺陷挖掘[4]和测试用例生成[5]。

符号执行技术面临的主要问题是分析的对象包含大量的路径分支，求解工具不能在有限的时间内完成穷举过程，即存在路径爆炸问题。如何有效地缓解路径爆炸问题是当前符号执行领域的热点之一，主要包括搜索约束和搜索策略这 2 个方面的研究：1) 搜索约束即对程序的输入进行建模[5-6]，或以目标位置为导向排除无关路径[4]，这类方法是通过约束对搜索路径进行裁剪，从而达到降低复杂度的目的；2) 搜索策略除了传统的深度优先或广度优先之外，还包括随机路径搜索和覆盖率优化等新的搜索方式，比如符号执行工具 KLEE[7]所用的算法就可以选择 4 种策略或这 4 种策略混合使用。本文方法是

收稿日期：2018−04−23；**修回日期**：2018−08−31
基金项目：中国科学院战略性先导科技专项基金资助项目（No.XDA-Y01-01）
Foundation Item: Strategic Priority Research Program of Chinese Academy of Sciences (No.XDA-Y01-01)

对符号执行工具进行改进，属于搜索约束方向。

搜索约束面向特定类型的程序或者目标时，能够大幅地裁剪路径，例如基于文档辅助对输入建模[6]，该方法从帮助文档、代码注释及文件格式中提取更多的搜索约束，帮助手册格式（manual）的参数约束以及特殊的文件格式，如可执行和可链接格式（ELF, executable and linking format）。然而该方法也很有局限性，比如需要程序带有帮助手册，同时符合解析的规范。如果一个程序缺少必要信息，比如缺少帮助文档，则无法使用该方法。所以该方法很难适用于所有的开源程序，尤其是一些缺乏维护的开源程序。

针对这个问题，本文提出了一种新的基于输入约束的符号执行（ICBSE, input constraint based symbolic execution）优化框架，通过提取代码中有效的命令行约束参数（比如 rm 命令中的-r 选项），从而实现对搜索路径的裁剪。本文贡献如下。

1) 面向开源代码，使用静态分析等方法获取输入命令行种类和长度以及命令行参数长度等必要约束信息，提升符号执行效率。

2) 使用源代码生成的中间语言低级虚拟机（LLVM, low level virtual machine）[8]作为提取对象，可以做到针对 LLVM 支持的平台和语言。

3) 基于开源工具 KLEE 实现 ICBSE 的支持，在增强了搜索约束的同时，还复用了 KLEE 已有的各类搜索策略。

本文使用 Linux 平台的常用工具 coreutils、grep、binutils、patch、diff 作为测试对象，实验结果表明，相比于 KLEE 原有的方案，ICBSE 方案在覆盖率上有约 20%的提升，在搜索时间上有约 15%的提升，同时该工具还发现了 4 个原工具未发现的缺陷。

2 相关工作

输入约束作为符号执行优化的方向之一，近年来有许多相关的研究工作。构建输入约束的方法包括手动构建、基于摘要产生约束、基于规范、基于运行时信息、基于文档等方法。

手动构建方面，文献[9]提出了称之为切片符号执行的方法。利用这种方法，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。该文献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，则会更加适用于实际程序的分析。

基于摘要产生约束方面，比如通过避免执行程序中的特定代码块来缓解路径爆炸问题。具体实现方法之一就是将需要重复执行的代码块，如函数在第一次运行时生成摘要，在下次执行该代码块时直接使用该摘要而不用重复执行该代码块，该方法有效地缓解了代码块因调用次数增多和调用深度增加引起的路径爆炸问题。摘要一般由代码块的前置条件和后置条件的析取构成。文献[5]利用过程内分析方法对函数生成摘要来辅助符号执行的过程间分析，有效缓解了因调用函数次数过多带来的路径爆炸问题。

基于规范方面，比如文献[10]为字符串指定规范，文献[11]为程序中的正则表达式指定规范。

基于运行时信息方面，Ramos 等[12]提出了对符号“惰性初始化”的方法，也就是前文提到的长度约束。文献[12]所提方法的思想是当为每个数据结构（特别是复杂数据结构）建模，在声明或者定义时只为其构建类型信息，直到被使用的时候，才根据使用的需要来初始化该变量的对象信息。这种方法实际上是先运行程序，再根据运行过程中对符号值做的运算决定符号值的长度约束，例如对数组进行操作。但这种方法一般只对程序当前执行路径上的变量长度约束提取有效果，而对于以命令行选项/参数为输入类型的测试效果并不明显，远远达不到约束收集的要求，除非将可能的选项都运行一遍。另外，这种方法对内容约束没有任何效果。

基于文档方面，Wong 等[6]提出了基于文档辅助的建模方法，即前文提到的内容约束。该方法本质上也是基于规范，但是约束性和导向性更强。所谓文档辅助即基于帮助文档、代码注释以及文档布局提取更多的输入约束，比如帮助手册格式的参数约束以及特殊的文档格式（比如 ELF）。这类约束与程序本身密切相关，能够大幅裁剪执行路径，但面临的问题也是显而易见的，比如需要程序带有帮助手册，同时符合解析的规范，并且程序处理的目标必须是工具支持的特定类型，否则该方法的作用不大。本文的工作是对基于文档方法的进一步改进，只需要源码就可以做到输入约束的收集。

3 背景介绍及示例

3.1 符号执行简介

符号执行指的是使用符号化的变量代替实际的输入，并将程序中代码行为转换为相应的程序状态

表达式的操作。程序状态包括变量的取值、路径条件（PC, path condition）和程序计数器。当遇到分支指令时，程序会分别搜索每个分支，并且将分支需要满足的条件加入路径条件中。程序的可达性通过约束求解器（constraint solver）对路径条件的可满足性进行求解来判断，从而生成路径对应的测试用例。

图 1 是一个简单的示例程序。其中，函数 foo() 包含了 2 个整型变量参数 x 和 y。该程序的符号执行过程如图 2 所示，执行树的初始状态是 S_0，其中，X 和 Y 分别表示 x 和 y 的符号值，该节点的路径条件为 true，类似的状态 S_1 和 S_2 的路径条件也为 true。当执行到 4(x>0)时，路径会分叉为 2 条路径，分别为状态 S_3(x>0)和 S_4(x≤0)。同时，到达 2 个状态的分支条件也会加入路径条件中，以判定路径的可行性。根据求解器的结果，表示存在相应的路径条件为 true 的 X 和 Y 的取值，随后生成可以到达状态 S_3 和 S_4 的测试用例。

```
1. int foo(int x, int y) {
2.     y++;
3.     x=x+y;
4.     if(x>0)
5.         x=x−1;
6.     else
7.         x=x+1;
8.     return x;
9. }
```

图 1　示例程序 1

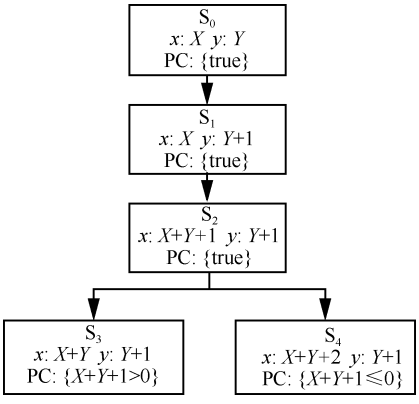


图 2　程序 1 的执行树

如上所述，目标程序将被分解为执行树，每个叶节点都对应了程序的一条执行路径，而每个分支节点则对应了程序的分支条件。随着程序分支变多，执行树的规模呈指数级增长，因而遍历整个执行树会引发状态爆炸问题。下面以一个更加复杂的例子说明本文提出基于输入约束的原因。

3.2　基于输入约束的优化示例

图 3 是一个简单的带参数的 C 语言程序。假设这个程序带有一个参数为“AAAAAAAAAA”，在满足输入时会进入相应的处理程序。

```
1. int main(int argc, char *argv[]) {
2.     int count=0;
3.     for(int i=0; i<10; i++) {
4.         if(argv[1][i]==“A”)
5.         count++;
6.     }
7.     if(count==10) {
8.         process_valid_input()     //bugs!!
9.     }
10. }
```

图 3　示例程序 2

图 3 中第 3 行和第 4 行的代码中是一个循环 10 次的比较语句，如果将循环展开的话，即 2^{10} 个分支。如果对输入没有任何限制的话，则只有 $\frac{1}{2^{10}}$ 的可能执行第 8 行对输入的操作以及进一步的操作。如果通过分析知道该程序需要参数 10 个“A”，则可以通过设置该参数到达第 8 行有缺陷的位置。同时发现该缺陷的概率由 $\frac{1}{2^{10}}$ 增加到 $\frac{1}{2}$。

以上描述说明能够快速发现缺陷的 2 个先决条件是：1) 符号化值的长度必须大于或等于 10；2) 符号化值的内容限制为某些值，比如示例中的 10 个“A”。条件 1)是必要条件，否则求解器是不可能解出长度超过 10 的结果；条件 2)是为了提高求解速度，对搜索空间进行了限制裁剪。下文将条件 1)称为“长度约束”，条件 2)称为“内容约束”。对于学术界流行的符号执行工具 KLEE 来说，是无法满足以上 2 个条件的，原因如下：1)KLEE 的符号长度是手动设置的，只能凭“经验”进行针对性的设置，而实际操作中，如果设置过短则无法触发该缺陷，如果设置过长则会造成搜索空间过大，搜索效率很低；2) KLEE 虽然提供内容约束的接口，但是不支持自动提取内容约束，使用者需要按照帮助文档将所有的内容约束加入约束集中，在实际操作中非常烦琐并且效率很低。

如何改进 KLEE 并将该方法推广到实际应用程序中是本文的主要研究点。在实际程序中，命令行参数是一种特殊的输入，用于触发某些功能，比如 rm 命令如果加上“-r”参数则表示以递归的方式删除文件/文件夹。在程序的代码实现中，通过 if-else

或 switch-case 语句检查合法参数，并进入该参数对应的处理流程。如果能够利用静态分析等方法定位到命令行处理流程的入口，并扫描各处理分支提取的选项内容以及参数长度，即可自动完成所有的提取工作。

基于以上基本思路，为了克服 KLEE 的缺陷，本文提出了 ICBSE 框架。使用该框架替换 KLEE 的前端，即自动从程序的源代码中提取所有的预设选项并作为输入约束驱动 KLEE 的符号执行引擎完成求解过程。比如 rm 命令有 15 个有效选项，ICBSE 会创建 15 个分支执行树。通过这种方式使所有的有效选项出现相同深度的执行树（假设有效选项为 -a、-b、-c、…、-o）。同时输入参数的长度也会作为 ICBSE 的提取目标之一加入约束集中。

以内容约束为例，如果不使用 ICBSE，那么测试-o 选项所需要的时间应该为总测试时间的 $\frac{1}{2^{15}}$，如图 4(a)所示。与之对比的是，如果开启了 ICBSE，则降为 $\frac{1}{15}$，如图 4(b)所示。总之，使用该方法对于程序的模型检测有 2 个方面的改进：1)在搜索空间中剔除了无效输入引入的搜索损耗，而集中检测有效的参数输入，从而将搜索集中在程序的核心功能代码区域；2)有些约束因为输入约束而简化，比如使用了实际的有效值，从而减少了约束求解器的计算时间。

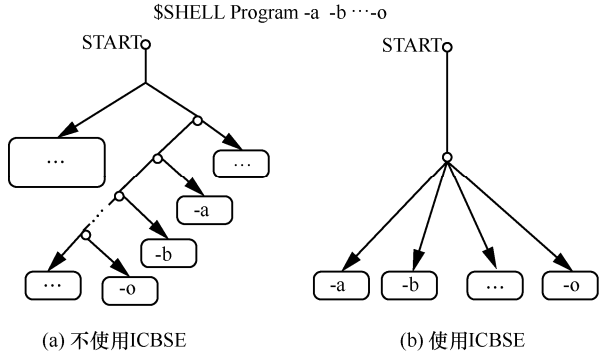


图 4　使用 ICBSE 前后执行树的变化情况

4　基于输入约束的符号执行优化

4.1　ICBSE 整体架构

ICBSE 整体架构如图 5 所示。本文选用 LLVM 的 bitcode 作为中间语言，利用提供的接口静态分析提取信息，方便和 KLEE 整合。随后对提取的信息进行筛选和转换，生成对应的约束格式。通过修改 KLEE 的符号生成器部分可以方便地读取已经生成约束格式继而生成对应的符号值，随后利用 KLEE 的符号执行引擎生成缺陷报告。

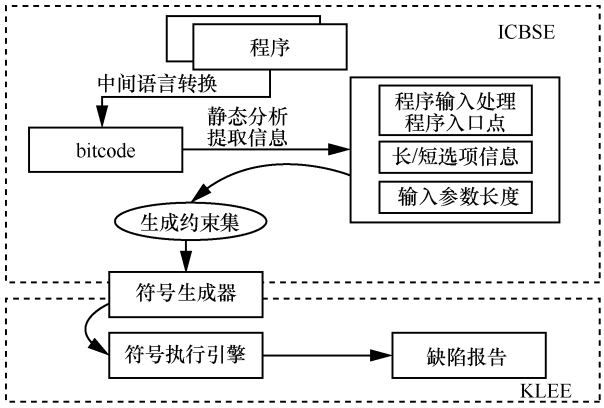


图 5　ICBSE 整体架构

4.2　命令行选项和参数

几乎所有的 Linux 程序都遵循一些命令行选项定义的约定。程序希望出现的参数可以分成 2 种，选项（option）和其他类型的参数。选项修饰了程序运行的方式，其他类型的参数则提供了输入（如输入文件的名称）。下面以使用 Linux 的 readelf 为例，说明提取的输入约束的内容，包括选项和参数 2 类，如图 6 所示。

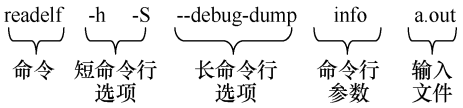


图 6　readelf 命令示例

图 6 中，“-h”和“-S”单横杠开始的为短命令行选项，这种选项后面不会跟任何参数，而“--debug-dump”这类双横杠开始的为长命令行参数，后面可能会跟不等长的命令行参数，比如本例中的“info”字段。最后还有可能跟一个文件名参数指示目标或源文件位置。

因为本文的工作是基于开源软件 KLEE 实现对 ICBSE 的支持，以上约束都需要转化为 KLEE 可接受的输入。KLEE 支持的选项包括：1)--sym-args，后面跟 min、max、len 这 3 个参数，表示的含义为 min-max 数量的符号参数，符号化长度为 len；2)--sym-file，后面跟 num、len 这 2 个参数，表示的含义为符号化文件的数量为 mum，长度为 len。图 6 中 readelf 命令行示例中的选项从左到右，有以下对应关系，如表 1 所示。

表 1　命令行选项和 KLEE 参数的对应关系

命令行输入	对应 KLEE 参数	说明
-h -S	--sym-args 0 2 2	readelf 带有 0~2 个长度为 2 的短选项
--debug-dump	--sym-args 0 1 13	readelf 带有 0~1 个长度为 13 的长选项
info	--sym-args 0 1 10	--debug-dump 命令行选项后带的命令行参数
a.out	--sym-files 1 20	符号文件数量为 1，长度为 20

从表 1 可以看出，KLEE 存在 2 个不足：1)KLEE 并不支持内容约束，比如长/短命令行选项，只能限制长度；2)KLEE 的长度约束都是基于经验手动设置的，比如表 1 中的命令行参数除 info 之外，还包括了其他如“abbrev”“pubnames”“ranges”之类的参数，其中最长为 13。

4.3　基本定义

本文选用了 LLVM 的 bitcode 作为中间语言，其中间语言自顶向下由模块（module）、函数（function）、代码块（block）、指令（instruction）这 4 个层次组成。其中，模块包含了函数，函数又包含了代码块，代码块又由指令组成。

指令再细分又包含了操作数（operand），操作数可以是值（val），比如算术运算，也可以是函数（func），比如函数调用。

本文在 bitcode 结构基础上定义了以下基本操作。

操作 1　B (f targetfunc)={block∈BC|∃Ins∈block, Ins.operand=targetfunc}

其中，BC 表示目标程序即 bitcode 的缩写，block 表示基本块，Ins 表示指令。操作 1 表示返回所有调用了目标函数 targetfunc 的基本块，且为了简化表示，所以没有按层次遍历。

操作 2　P (t type, b block) = {Ins.val | Ins∈block, Ins.itemtype=type}

其中，block 表示基本块，type 表示指令的类型，比如 switch、branch 等。操作 2 返回基本块中指令类型等于 type 的操作数（不包括函数）。

另外，在建立起长短选项的映射关系后，还会有以下操作和数据结构。

1) shortopset：短选项约束集。

2) longopset：长选项约束集。

3) arglenset：参数长度约束集。

4) hasargs(val)：返回某个选项是否带有参数。

5) arglen(val)：返回某个选项带有参数长度。

各选项集的建立过程请参见 3.4 节中的说明。

4.4　选项提取规则

本节将结合实际代码实现说明 ICBSE 的选项自动提取规则。在 Linux 程序实现中，按照标准都是通过调用 getopt_获取输入的参数。

规则 1　（内容约束 1）

∃ block (block ∈ B(getopt)) => add2shortopset (P(switch, block))

规则 1 表示任何代码块中只要在入口点调用了 getopt 函数，即作为起始点在代码基本块中收集约束。因为比对输入都是通过 switch-case 语句实现的，所以将基本块中所有 switch 类型的指令的值加到短约束集中。

以上过程以 readelf 的源码进行简单的说明，代码示例如图 7 所示。

```
//readelf.c 2629-2787
1. while ((c=getopt_long (argc, argv,
        "……",
        options, NULL)) != EOF)
2. {
3.      char *cp;
4.
5.      int section;
6.      switch (c)
7.              {
8.                case 0:
9.                      /* Long options. */
10.               break;
11.               case 'H':
12.                   usage ();
13.               break;
14.              .....
15.         }
16. }
```

图 7　readelf 代码示例

通过扫描目标代码块发现，第 2 行调用了 getopt_long 函数（包括 getopt）的输入参数，即表示从该位置开始搜集选项。遍历该代码块中的每条比较语句，即代码中的 case 语句，表示对选项进行处理，比如第 10 行的 case 'H'表示该行操作数“H”可以生成短选项，生成如“-H”类型的选项。依次扫描 case 语句的操作数，可生成如“-e”“-r”“-s”等短选项约束。

规则 2　（内容约束 2）

∃ val (val∈longopset) and (val∉shortopset) =>add2longopset(val)

规则 2 的作用是如果长选项没有对应的短选项，则将该长选项加入约束集中。使用规则 2 可以

适当地过滤功能相同的输入选项，减少搜索的路径。因为程序通常会提供包括长选项和短选项这 2 种选项形式的参数，通过规则 1 已经生成了短选项约束集 shortopset，而长选项约束集 longopset 以及长/短选项之间的映射关系可以由扫描 option 结构体获取，表 2 列出了部分的映射关系。

表 2　　长短选项映射关系

短选项	长选项	是否需要参数
-a	--all	no_argument
-h	--file-header	no_argument
-D	--use-dynamic	no_argument
-x	--hex-dump	required_argument
无	--debug-dump	optional_argument
-v	--version	no_argument

比如，“-a”和“---all”指向的是同一类功能的选项，那么规则中需要将短选项“-a”加入输入约束即可。而第 5 行的“--debug-dump”没有对应的短选项，则需要单独加入输入约束。值得注意的是，第 3 列表示该选项是否需要参数，比如 required_argument 表示必须参数，而 optional_argument 表示可选参数。

规则 2 是对内容约束的一次过滤，同时也会记录下哪些选项需要紧跟参数，用于规则 3 的生成。

规则 3　（长度约束 1）

∃ val (val∈shortopset and hasargs(val)) or (val∈longopset and hasargs(val)) => add2arglenset (arglen(val))

规则 3 的作用在于收集长/短选项自带的参数长度，并将该约束加入约束集中。规则 3 的前提是已经通过规则 1、规则 2 以及代码中的关键数据结构生成了短选项约束集和长选项约束集，并完成了参数长度的收集工作。在此基础上收集某个选项详细处理过程，比如前面提到的长项选项“--debug-dump”，通过静态分析代码可知，该选项可以带“line”“info”“abbrev”“pubnames”“ranges”“macro”“frames”“frames-interp”“str”“loc”等附加参数。该条规则会选择最长的参数“frames-interp”并将其长度 13 加入约束集中。但规则 3 并不将以上参数作为内容约束加入约束集中，原因是有些长选项可以跟不定内容的参数，比如“--hex-dump”后面跟的是目标文件名，所以该条规则被称为长度约束规则。

综合以上 3 条提取规则，整个选项提取过程算法如算法 1 所示。如前所述，目标代码为 bitcode 格式，所以简称为 BC。同时为了简化，算法的循环中省略了模块和基本块的遍历，只描述函数和指令的遍历。

算法 1　约束提取算法

输入　bitcode 格式的目标代码

输出　约束集

1) constraints set C = Ø;//约束提取算法
2) input set in;
3) for each func $F_i \in$ BC do
4)　　for Inst $I_j \in F_i$ do
5)　　if I_j call getopt_long
6)　　shortvars= shortvars∪getshortoptions (I_j);
7)　　shortvarNeedArg=shortvarNeedArg∪getshortoptionsArgInfo(I_j);
8)　　longvars=longvars∪getlongoptions (I_j);
9)　　longvarNeedArg=longvarNeedArg∪getlongoptionsArgInfo(I_j);
10)　　end if
11)　　if I_j have case with option
12)　　get the option index k in shortvars or longvars
13)　　in$_k$.op = in$_k$.op∪option;
14)　　if(shortvarNeedArg[k]=true or longvarNeedArg[k]=true)
15)　　get the length of the argument L;
16)　　in$_k$.arglen=L;
17)　　end if
18)　　end if
19)　end for
20) end for
21) //base on the rule1~3 to add constraints to C;
22) return C

算法 1 沿用了 LLVM 提供的中间代码处理接口，比如 F 和 I 分别表示函数体和函数体内的指令。算法中第 6)~9)行搜集所有的长/短选项，并确定该选项是否带有参数信息，其中第 6)行应用的是规则 1，而第 8)行应用的是规则 2。而第 12)~17)行表示如果选项带有参数，则监控对该选项读取的参数处理的语句，并搜集所有的长度，也就是规则 3 的实现。算法 1 按照规则 1~规则 3 将以上搜集的信息转换为 ICBSE 可接受的格式加入约束集中，这里使用了

KLEE 提供的 assume 函数作为约束输入接口，在此不再展开描述。

5　实验与分析

本文基于符号执行工具 KLEE 实现了 ICBSE 优化框架的支持。为了方便对比，本文的搜索策略都选择随机搜索，并在此基础上展开了一系列的实验，主要从 3 个方面对比 KLEE 的检测结果：1) 缺陷检测能力，包括发现的缺陷数量和缺陷类型；2) 代码覆盖率，包括指令覆盖率和分支覆盖率；3) 执行时间。

本文选用了 Linux 常用工具作为检测目标，这些工具在之前的研究中也曾做过相关的测试[12-16]。工具的名称、版本以及代码行数如表 3 所示。

表 3　　测试对象

名称	版本	简介	代码行数
coreutils	6.11	Linux 常用工具集合，本文选用 ls 和 sort 这 2 个常用程序作为测试对象	ls:3 241 sort:2 317
binutils	2.14	Linux 二进制工具集合，本文选用 readelf 作为测试对象	12 076
diffutils	2.8	Linux 文本比较工具，本文选用 diff 作为测试对象	1 114
grep	3.0	Linux grep 命令	6 144
sed	4.2	Linux Sed 命令	4 125
patch	2.7	Linux 补丁工具	5 996

所有实验都是在处理器为 Intel Xeon X5675 CPU（24 core，3.07 GHz）、内存为 94 GB 的服务器上进行的，操作系统是 64 位的 Ubuntu 14.04 LTS。KLEE 的主要参数设置如下：--max-time=3 600（最大执行时间为 3 600 s，即 60 min），--max- solver-time=80（最大求解时间为 80 s），--search=random（设置搜索策略为随机）。

5.1　缺陷检测能力

通过一系列实验，KLEE 发现了 3 个未知的错误，而使用 ICBSE 优化后发现了 7 个未知的错误（其中 4 个 KLEE 并没有检测到）。表 4 展示了检测的结果。

以 patch 发现的缺陷为例说明 ICBSE 的检测能力。首先，“-d”作为短选项已经加入约束集中，所以很快就能触发对应的路径搜索，如图 8 的代码所示。其中，第 7~8 行表示命令行选项为“-d”同时后面跟踪命令参数 arg 时，patch 会将工作目录切换到 arg 指示的文件夹位置。第 9 行以及第 14~15 行表示如果 arg 指向的位置无效会产生错误退出，退出之前会清空一些临时数据。第 20 行表示随后清空流程会进入输出文件夹清空输出文件，但此时由于切换目录失败，因此 file_to_output 参数为空指针，从而产生了空指针错误。在 patch2.7 版本中使用“patch -d -”命令即可重现该错误。而 KLEE 在达到最大的运行时间后并未检测出该缺陷。从结果可以看出，因为 ICBSE 能够更有效率的构造合法的输入，所以能够搜索更多功能实现相关的代码，进而更有效地发现核心功能的缺陷。其他的缺陷也已经反馈给开发者做了确认和相应的修改。

表 4　　测试结果

程序	位置	错误类型	KLEE	ICBSE
objdump	elf-attrs.c:463	整型溢出		√
readelf	readelf.c:532	指针越界	√	√
readelf	readelf.c:712	指针越界	√	√
readelf	readelf.c:2662	空指针	√	√
patch	gl_list.c:215	空指针		√
bool	text.c:231	空指针		√
head	head.c:297	内存耗尽		√

```
//patch.c
1. while ((optc = getopt_long (argc, argv, shortopts, longopts, (int *) 0))!= −1) {
2. switch (optc) {
3.       //other options
4.       case 'c':
5.          diff_type = CONTEXT_DIFF;
6.       break;
7.       case 'd':
8.         if (chdir(optarg) < 0)
9.           pfatal ("can't change to directory %s", quotearg (optarg));
10.      break;
11.      case 'D':
12.        //other options
13. }
14. void fatal_exit (int sig)
15. {
16.       cleanup ();
17.       //other operations
18. }
19.
20. static void cleanup (void)
21. {    //other operations
22.       forget_output_files ();
23. }
24. static void forget_output_files (void) {
25.       gl_list_iterator_t iter =
gl_list_iterator (files_to_output);//空指针
26.      //other operations
27. }
```

图 8　　patch 中的错误代码

5.2 代码覆盖率

在代码覆盖率测试中，本文关注了指令覆盖率和分支覆盖率这2个指标。指令覆盖率就是度量被分析代码中每个可执行语句是否被执行到了，分支覆盖率也是类似。本文使用 gcov 作为统计语句覆盖率的工具。值得注意的是，统计覆盖率的对象是程序本身的可执行代码，而不考虑各程序链接进来的程序库的代码。本文对比了 ICBSE 和 KLEE 的执行结果，如表5所示，其中，IC 表示指令覆盖率，BC 表示分支覆盖率。

表5 代码覆盖率

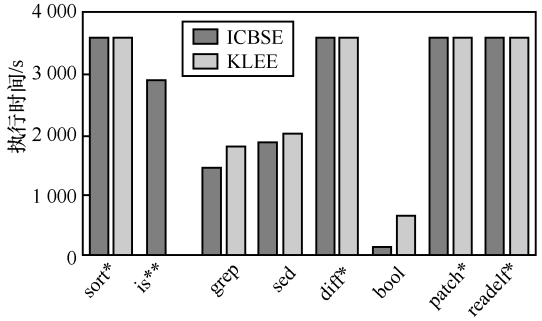
程序	ICBSE		KLEE		对比结果	
	IC	BC	IC	BC	IC	BC
sort	34.78%	24.15%	28.23%	19.81%	44%	22%
ls	40.31%	28.79%	30.81%	22.10%	40%	30%
grep	25.58%	16.29%	22.75%	14.46%	12%	13%
sed	29.35%	18.17%	23.27%	14.83%	26%	23%
diff	26.71%	20.03%	20.86%	15.94%	28%	26%
bool	25.19%	17.22%	20.56%	13.31%	23%	29%
patch	28.43%	20.27%	20.67%	15.44%	38%	31%
readelf	20.36%	15.03%	20.66%	15.26%	−1%	−1.5%

在代码覆盖率的对比测试中，ICBSE 和 KLEE 均采用同样长度的输入参数。从结果来看，除 readelf 外，其他程序的指令覆盖率和分支覆盖率都有明显的提升。以 diff 程序为例，diff 程序有 55 个输入选项，通过手动检测，KLEE 只搜索了 55 个选项路径中的 27 个，而使用 ICBSE 后提升到了 53 个（全部 55 个），所以指令覆盖率和分支覆盖率分别提升了 28%和 26%。值得注意的是，ICBSE 并未搜索全部的选项，因为在实际操作中，设置了长命令行选项的阈值为 20，所以有 2 个超长（长度≥20）的命令行选项，如果选择覆盖的话，对比测试中会导致工具迅速耗尽内存。

另外，readelf 对比基本持平，无明显提升。其主要原因在于 readelf 除了命令行选项的约束外，还受特定的文件格式约束，可以看出，在面向特定格式的程序中，输入的文件内容对符号执行的覆盖率影响更大，这也是在后续的优化中需要继续关注的。

5.3 执行时间

执行时间的对比测试与覆盖率对比测试环境一致，最大执行时间限制在 3 600 s，即 60 min，如果超过 60 min 则自动结束。执行结果如图 9 所示，其中，深色为 ICBSE 的运行时间，浅色为 KLEE 运行时间，其中有一半的程序在限定时间内完成了测试。从结果来看，需要注意 3 点：1) sort、diff、patch、readelf 执行时间统一为 3 600 s，即达到了最大的执行时间，但是如果考虑到限定之间内达到的代码覆盖率，ICBSE 的执行效率好于 KLEE；2)完成测试的 grep、sed、bool，ICBSE 所用的运行时间明显短于 KLEE；3) ls 的 KLEE 测试部分因为内存耗尽而终止了运行，所以没有结果。根据以上结果，结合相同执行时间内代码覆盖率的提升效果，可以得出 ICBSE 在执行时间上优于 KLEE 的结论。



注：*表示 ICBSE 和 KLEE 在限定时间内都没有结束；
**表示 KLEE 因为内存耗尽而没有显示执行时间。

图9 执行时间对比

6 结束语

本文提出并实现了一种基于输入约束的符号执行优化。实验表明，相比于原有的方案，使用 ICBSE 优化框架后在缺陷查找能力上有明显提升，并发现了 4 个 KLEE 方案不能发现的缺陷，同时在覆盖率上有约 20%的提升，在搜索时间上有约 15%的提升。

后续的研究工作主要从 2 个方面开展。

1) 在前面已经提到，类似于 readelf 除了命令行选项的约束外还受特定的文件格式约束，在面向特定格式的程序中，输入的文件内容对符号执行的结果影响更大。后续将尝试使用惰性初始化的方法改进这方面的不足，如果输入文件采用标准结构的文件（如某个可执行文件），则将文件分为布局控制结构（如 ELF 的文件头）和内容数据这 2 个部分。

2) 当前的输入约束收集都是基于程序对输入选项和参数的处理过程进行收集的，而并未考虑程序中对于输入的使用和判定。后续可以使用过程间静态分析，结合数据流分析、别名分析等方法提取相关的输入变量在程序中的使用信息，从而搜集到

更准确的约束，进而对执行路径进行进一步的裁剪。

参考文献：

[1] 张健. 精确的程序静态分析[J]. 计算机学报, 2008, 31(9): 1549-1553.
ZHANG J. Sharp static analysis of programs[J]. Chinese Journal of Computers, 2008, 31(9): 1549-1553.

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

[3] CADAR C, GODEFROID P, KHURSHID S, et al. Symbolic execution for software testing in practice: preliminary assessment[C]// International Conference on Software Engineering. 2011: 1066- 1071.

[4] ANAND S, GODEFROID P, TILLMANN N. Demand-driven compositional symbolic execution[C]//Theory and Practice of Software, International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems. 2008: 367-381.

[5] GODEFROID P. Compositional dynamic test generation[C]//ACM Sigplan-Sigact Symposium on Principles of Programming Languages. 2007: 47-54.

[6] WONG E, ZHANG L, WANG S, et al. DASE: document-assisted symbolic execution for improving automated software testing[C]// IEEE International Conference on Software Engineering. 2015: 620-631.

[7] CADAR C, DUNBAR D, ENGLER D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//USENIX Conference on Operating Systems Design and Implementation. 2009: 209-224.

[8] LATTNER C, ADVE V. LLVM: a compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization. 2004: 75-86.

[9] DAVID T, ANDREA M, NOAM R, et al. Chopped symbolic execution[C]//The 40th International Conference on Software Engineering. 2018: 350-360.

[10] BJØRNER N, TILLMANN N, VORONKOV A. Path feasibility analysis for string-manipulating programs[C]//TOOLS and Algorithms for the Construction and Analysis of Systems, International Conference. 2009: 307-321.

[11] VEANES M, HALLEUX P D, TILLMANN N. Rex: symbolic regular expression explorer[C]//Third International Conference on Software Testing, Verification and Validation. 2010: 498-507.

[12] RAMOS D A, ENGLER D. Under-constrained symbolic execution: correctness checking for real code[C]//Usenix Conference on Security Symposium. 2015: 49-64.

[13] AVGERINOS T, REBERT A, SANG K C, et al. Enhancing symbolic execution with veritesting[C]//International Conference on Software Engineering. 2014: 1083-1094.

[14] MARINESCU P D, CADAR C. Make test-zesti: a symbolic execution solution for improving regression testing[C]//International Conference on Software Engineering. 2012: 716-726.

[15] CADAR C, GODEFROID P, KHURSHID S, et al. Symbolic execution for software testing in practice: preliminary assessment[C]// International Conference on Software Engineering. 2011: 1066-1071.

[16] MARINESCU P D, CADAR C. KATCH: high-coverage testing of software patches[C]//Joint Meeting on Foundations of Software Engineering. 2013: 235-245.

[作者简介]

汪孙律（1986− ），男，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。

林渝淇（1988− ），男，山东潍坊人，博士，中国科学院软件研究所助理研究员，主要研究方向为系统安全与可信计算。

杨秋松（1977− ），男，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，主要研究方向为软件工程、形式化方法、模型检测、安全操作系统。

李明树（1966− ），男，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，主要研究方向为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。