# 摘 要

近年来，计算机逐渐普及，进入千家万户。而在这些计算机上跑的程序必须要禁得起考验。如何检验一个程序是否满足规范性、安全性、可靠性、可维护性，是计算机科学界和工业界共同关注的热点。

程序分析方法按是否执行程序，可以分为静态分析技术和动态分析技术两类。静态分析技术中使用最广泛的技术之一是符号执行技术。符号执行技术以符号量作为输入，来模拟执行程序，求出所有路径的集合，并根据一条路径上的所有约束条件求解变量值。由于符号执行技术面临着路径空间爆炸的问题，为了提高可行性，出现了动态符号执行技术，以具体值作为输入。

本文首先介绍符号执行技术和动态符号执行技术的原理及其特点，而后介绍基于动态符号执行的分析工具klee。在klee的基础上，实现源程序分析工具，着重处理全局变量相关信息，并对各种类型的全局变量进行测试。

**关键词：**静态分析**，**动态符号执行，全局变量

# Abstract

In recent years，the increasing popularity of computers have been come into millions of households．The Program run on these computers must withstand the test．How to check whether a program satisfies a specification， safety，reliability，maintainability，is a hotspot of computer science and industry．

Program analysis method according to whether the execution of the program，can be divided into static analysis and dynamic analysis techniques two．In all static analysis techniques，one of the most widely used techniques is symbolic execution techniques．Symbolic execution technique use symbolic value as input，and simulate the execution of the program．During the simulation execution，it calculates the value of variables according to all the constraints on the path．And also，it will obtain all the possible paths．The symbolic execution technology is facing the path space explosion problem．In order to improve the feasibility，dynamic symbolic execution techniques appeared，it uses specific values as input．

This paper introduces the principle and characteristics of symbolic execution technology and dynamic symbolic execution technique，and then introduces the dynamic symbolic execution analysis tool KLEE．On the basis of klee，we design the analysis tool for source code．It focuse on global variables related information，and tested for various types of global variables．

**Keywords：**static analysis，dynamic symbolic execution，global variables

**目 录**

[摘 要 I](#_Toc390353404)

[Abstract II](#_Toc390353405)

[1 绪言 1](#_Toc390353406)

[1.1 课题背景 1](#_Toc390353407)

[1.2 课题的主要研究工作 1](#_Toc390353408)

[1.3 课题研究的目的和意义 1](#_Toc390353409)

[1.4 国内外概况 2](#_Toc390353410)

[2 相关技术 4](#_Toc390353411)

[2.1 符号执行 4](#_Toc390353412)

[2.1.1 基本原理 4](#_Toc390353413)

[2.1.2 符号执行示例 6](#_Toc390353414)

[2.1.3 符号执行面临的问题 6](#_Toc390353415)

[2.2 动态符号执行工具KLEE 8](#_Toc390353416)

[2.2.1 KLEE简介 8](#_Toc390353417)

[2.2.2 LLVM介绍 8](#_Toc390353418)

[2.2.3 内存模型 10](#_Toc390353419)

[3 源程序分析工具设计与实现 16](#_Toc390353420)

[3.1 需求分析 16](#_Toc390353421)

[3.2 系统设计与实现 16](#_Toc390353422)

[3.2.1 全局变量在中间文件中的表示 16](#_Toc390353423)

[3.2.2 具体实现 18](#_Toc390353424)

[3.3 测试分析 19](#_Toc390353425)

[4 总结与展望 20](#_Toc390353426)

[致 谢 21](#_Toc390353427)

[参考文献 22](#_Toc390353428)

[附录 测试程序 24](#_Toc390353429)

# 1 绪言

## 1.1 课题背景

随着计算机科学技术的飞速发展，特别是Internet出现以来，计算机迅速普及，进入各行各业，千家万户。软件中一个漏洞可能带来无法估量的损失，特别是在广泛使用的协议或者软件中。比如，OpenSSL“Heartbleed”安全漏洞[1]至少影响几十万的服务器的安全，很多国内外知名的网站的安全都受到威胁，影响到广大网民的隐私和财产安全。因此，软件的安全性越来越受到人们的重视，如何检测软件中存在的安全问题也逐渐成为软件工程领域的研究热点。

近年来，多线程程序设计技术已经广泛应用到软件开发中。同时，多线程技术的发展也带来了数据竞争的问题。线程之间的同步和互斥，是编写多线程程序时必须要考虑的。对于多线程程序的分析测试工作也比单线程程序要困难得多。针对源代码的静态分析技术，是一种有效的程序分析方法。

## 1.2 课题的主要研究工作

静态分析方法中，有一种常用的分析方法，符号执行。符号执行是以符号量代替程序中的变量值，并模拟执行程序，覆盖程序中的所有路径并根据路径上约束，求出变量取值的分析技术。而符号执行技术由于存在路径爆炸等问题[2]，实际使用时效率较低。为了解决这个问题，提高符号执行技术的可行性，P.Godefroid等人[3]将符号执行技术与动态分析技术想结合，提出了动态符号执行技术。本课题的主要研究工作就是使用动态符号执行技术，分析源代码文件，并给出源代码文件中全局变量每处调用的位置和读写类型。

## 1.3 课题研究的目的和意义

课题研究的意义：

（1）研究动态符号执行技术及相关的基础理论，并应用到程序分析中；

（2）给出全局变量的调用信息，为多线程时序分析和数据竞争检测提供便利。

## 1.4 国内外概况

程序分析技术有静态分析技术和动态分析技术两类。

静态分析技术是指在不执行程序的情况下，对程序源代码进行分析，验证代码是否符合规范性、安全性、可靠性、可维护性等要求的一种程序分析技术。静态分析工具一般包括预处理器、数据库、错误分析器、报告生成器四部分[4]。林阿龙[5]利用程序静态分析技术，设计了一个C源程序分析工具CAAT。常用的静态分析技术的实现方法有模型检验、数据流分析、抽象解释、谓词转换、定理证明、类型推导、符号执行等。

动态分析技术是通过实际地执行程序，检测程序执行过程中的状态、结果等。既有传统的测试人员手工测试，也有使用自动化测试工具进行测试的。如，IBM的Rational PurifyPlus[6]就是一个使用动态分析技术的测试工具，并且投入商业化使用。动态分析技术与静态分析技术相比，它关注的是程序的功能和表现，只能检测出实际存在的缺陷，但它更符合生产环境的需要。静态分析技术能够对程序中所有可能执行到的路径进行检测，能够发现潜在的缺陷，但是误报率高，容易将一些正确代码定位为缺陷。可以说，静态分析技术与动态分析技术各有所长，两者结合使用可以取长补短，更好的排除程序中的错误。

静态分析技术中非常重要的一种方法是符号执行。符号执行于1976年由James C．King[7]首先提出，经过三十多年的发展，至今仍然被广泛研究。符号执行技术在软件测试和程序验证中发挥着重要作用。符号执行是指在不执行实际程序的前提下，把源程序翻译为一种中间语言，用符号值表示程序变量的值，然后基于中间语言模拟程序执行来进行相关分析的技术。目前，符号执行技术被广泛应用在程序分析检测[8]和测试数据生成[9]中。

Cristian Zamfir和George Candea使用符号执行技术实现了一个用于软件自动Debug的工具[10]。只需要输入程序和错误报告，就能自动找到导致这个错误的执行路径。这为调试一些难以重现的bug提供了极大的便利。

由于符号执行存在路径空间爆炸等问题，结合动态分析的特点，P.Godefroid等人[3]提出了动态符号执行技术。动态符号执行与传统静态符号执行的区别在于输入值的表现形式不同，动态符号执行以具体值作为输入来模拟程序执行。由于动态符号执行使用具体值，模拟执行过程中的开销比使用符号值的符号执行要小很多。但是，也正是因为这个原因，动态符号执行产生的路径集合比符号执行要小，是一个近似的路径集合。

目前，已经有许多动态符号执行测试工具，大致可以分为三个系列。

P．Godefroid等人开发的DART[3]是最早的动态符号执行测试工具之一。它的测试目标包括了程序崩溃、断言失败等标准错误，只支持整数类型约束，用随机测试来处理遇到含有指针类型约束的情况。SMART[11]是基于DART开发的测试工具，在DART上增加了一个效率更高的搜索算法，目标是使组合测试输入生成效率提高。基于DART，P．Godefroid等人还开发了SAGE[12]，用于测试机器码的动态符号执行测试工具，该工具目前不支持指针解引用的符号化。SAGE在针对函数指针的处理上，增加了高阶测试输入数据生成的想法，有初步试验证实其有效性。

K．Sen等人开发的CUTE[13]，CREST[14]都是用于C程序测试的动态符号执行测试工具。CUTE不能处理含有符号偏移(输入变量作为偏移)的约束。CREST是基于CUTE之上做的扩展。CREST不支持函数指针调用，但支持位移运算(不支持位掩码)。实证研究[14]表明了CREST在1000-2000行代码的单元测试中的有效性。jCUTE[15]是K．Sen等人开发的用于测试Java程序的测试工具。

C．Caciar等人开发了EXE[16]，KLEE[17]和Rwset[18]，它们是另外一系列的动态符号执行测试工具。这三个测试工具也都是用于C程序测试的。KLEE基于EXE，为了应对路径爆炸的问题，对程序与外部环境的交互做了额外处理。和CREST—样，EXE也不支持函数指针。不同于DART和CUTE，KLEE可以处理含有指针符号、偏移符号的约束。Rwset也是对路径爆炸的问题做了处理，主要增加了对不必要的路径进行剪枝的算法。

现有的动态符号执行测试工具中，除了上述三个系列之外，还有许多针对面向对象编程语言的测试工具。比如，jFuzz[19]，它是K．Jayaraman等人基于JPF[20]开发的。jFuzz用于java程序测试，它能支持位运算。PEX[21]则是N．Tillman等人基于.Net平台开发的，用于测试C#程序。由于java和C#都没有指针类型，因此无需考虑指针约束。

# 2 相关技术

## 2.1 符号执行

### 2.1.1 基本原理

符号执行是指在不执行实际程序的前提下，把源程序翻译为一种中间语言，用符号值表示程序变量的值，然后基于中间语言模拟程序执行来进行相关分析的技术。

首先介绍一些相关概念。

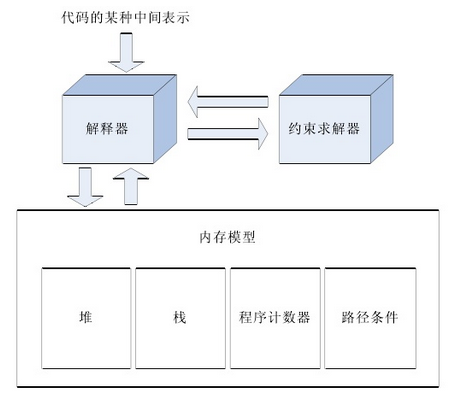


图2.1 符号执行系统整体结构

程序的路径（path）是程序的一个语句序列，这个语句序列包括程序的一些顺序的代码片段，代码片段之间的连接是由于分支语句导致的控制转移。一个路径是可行的（feasible），是指存在程序输入变量的至少一组值，如果以这组值作为输入，程序将沿着这条路径执行。否则，路径就是不可行的（infeasible）。路径条件（path condition，PC）是针对一个路径的，它是一个关于程序输入变量的符号值的约束，一组输入值使得程序沿着这条路径执行当且仅当这组输入值满足这条路径的路径条件。图2.1是符号执行系统的整体结构示意图。

符号执行分为过程内分析和过程间分析，过程间分析又叫全局分析。过程内分析是针对单个过程中的代码进行分析；过程间分析指对整个程序代码进行上下文敏感的分析。所谓上下文敏感分析是指在当前函数入口点要考虑当前的函数间调用信息和环境信息等。程序的过程间分析是在过程内分析的基础上进行的，但过程内分析中包含了函数调用时就也引入了过程间分析，因此两者之间是相对独立又相互依赖的关系。

过程内分析流程如图2.2所示。首先，对待分析的程序代码，构建控制流图（Control Flow Graph，CFG）。控制流图（CFG）是编译器内部用有向图表示一个程序过程的一种抽象数据结构，图中的节点表示一个程序基本块，基本块是没有任何跳转的顺序语句代码块，图中的边表示代码中的跳转，它是有向边，起点和终点都是基本块。在CFG从入口节点开始模拟执行，在遇到分支节点时，使用约束求解器判定哪条分支可行，并根据预先设计的路径调度策略实现对该过程所有路径的遍历分析，最后输出每条可执行路径的分析结果。其中，约束求解是数学上的判定过程，形象地说是对一系列的约束方程进行求解。

如果要进行源代码的安全性检测，则需要在过程内分析时，根据具体的规则知识库来添加安全约束。例如，如果要添加缓冲区溢出的安全约束，则在执行时遇到对内存进行操作的语句时，就要对该语句所操作的内存对象的边界添加安全约束。以上面的方式来进行安全约束的添加，并且每次在添加之后就使用约束求解器对所有的安全约束进行求解，以判定当前是否可能潜在一个安全问题。



图2.2 过程内分析原理流程图

程序过程间分析流程如图2.3所示。首先，为整个程序代码构建函数调用图（Call Graph，CG），在函数调用图中，节点表示函数，边表示函数间的调用关系。根据预设的全局分析调度策略，对CG中的每个节点（对应一个函数）进行过程内分析，最终给出CG每种可行的调用序列的分析结果。



图2.3 过程间分析原理流程图

### 2.1.2 符号执行示例

如图2.4所示，是一段简单的示例代码的符号执行过程。这段代码只包含一个if分支。首先，对这段代码构建控制流图CFG，生成的CFG中只包含两条路径。然后，对CFG进行分析。使用符号值作为i的输入，模拟执行程序，在遇到分支语句if (i > j)时，对i产生了一个约束。对于i>j的路径，则对i有约束条件i>10；对于另一条路径，有约束条件i<=10。使用约束求解器判定这两条路径的可行性，在本示例中使用的是深度优先调度策略。在对两条路径进行模拟执行和约束收集后，可以用约束求解器求出分别触发这两条路径的两个测试用例和对应的执行结果。

通过这个简单的例子，可以看出使用符号执行技术进行分析能够达到很高的路径覆盖率。结合约束求解器，可以实现测试用例的自动生成。

### 2.1.3 符号执行面临的问题

（i）路径空间爆炸

符号执行技术在理论上面临着路径状态空间的爆炸问题，其主要形成原因是每一个分支条件语句都可能会使当前的路径再分支出一条新的路径，而这是“指数级”增长的。

为了降低路径空间爆炸带来的影响，目前主要有两类做法。一是，在具体的实现中使用限定每个过程内的分析路径数目上限，来缓解该问题产生的影响；或者使用设置时间上限或内存空间上限的方法来缓解路径空间爆炸问题所可能造成分析工具崩溃的恶劣影响。二是，通过设计更好的路径遍历策略，以在有限的时间和空间范围内达到最大的代码检测覆盖率。人们通过改进路径调度算法来提高符号执行的分析性能，但是这些路径调度算法都只是局部改进，很难从根本上解决这一问题。

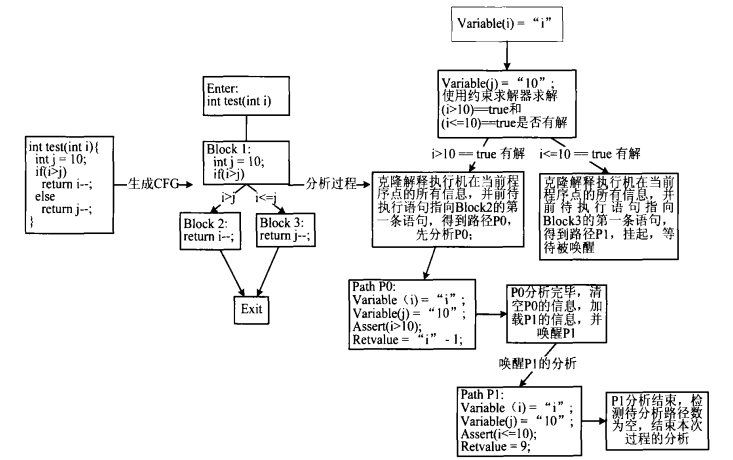


图2.4 符号执行示例

（ii）复杂结构语义和操作意义的建模

符号执行实现时需要对被分析代码的结构语义进行建模，然后再对被分析代码的操作语义进行建模，最后构建一个虚拟机模型。由于符号执行是路径敏感的分析方法，因此一般为每条路径都会创建一个专属的虚拟机模型，以保证路径之间的相互独立性。该模型的准确程度将直接影响静态分析结果的精度。由于编程语言中使用的复杂数据结构和复杂操作语句具有较高的灵活性，使得它们的建模变得十分困难。

Visser W等人[22]提出了一种惰性初始化的方法。其思想是为每个数据结构（特别是复杂数据结构）建模时，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。

（iii）程序全局分析

在程序全局分析过程中，当对一个规模较大、包含很多的过程间调用的程序进行上下文敏感的分析时，每当一个过程调用了另一个过程时都进入子过程进行分析，虽然会很精确，但这种方式可能会造成大量的时间空间花销，而使分析过程异常终止或在用户可接受的时间内无法完成。

一种比较好的全局分析方法叫做“函数摘要”。函数摘要的方法是在过程内分析的基础上对己分析过的函数进行一个摘要记录的操作。在以后的分析中遇到调用其他函数时，如果己存在被调用函数的摘要，则直接调用该函数的摘要并对该摘要行为进行解释；如果不存在被调用函数的摘要信息，则进入被调用函数进行分析，并在分析之后进行摘要保存。函数摘要是一种相对折中的办法，所创建的函数摘要也可能很不准确。另一个问题是对无法获得源代码的第三方库函数的摘要只能通过人工的编写来完成，这也可能是不精确的，而这两点都会影响到最终的分析结果的精度。

对于函数摘要的构建策略也是一个研究内容，Cadar C等人的文章[23]使用的是在函数调用图上的自顶向下的需求驱动的策略来为所有子过程创建函数摘要。犹豫循环语句可能会使被摘要函数的路径空间爆炸，而使得无法对函数的所有路径都进行分析摘要，该文章介绍了结合使用循环不变量来进行函数摘要以解决循环语句所引入的问题

## 2.2 动态符号执行工具KLEE

本节介绍符号执行工具KLEE，并用几个小程序进行测试。

### 2.2.1 KLEE简介

KLEE是一个符号执行工具，基于开源项目LLVM，为复杂系统软件生成的测试数据可以达到很高的覆盖率。同时，KLEE是一个错误检查工具，除了常规的编码错误，也可以检查功能错误。KLEE使用约束求解器STP，扮演着处理符号进程的操作系统和解释器的双重角色。KLEE把内存视为多个无类型字节数组，为每一个数据对象生成一个单独的字节数组。无类型字节数组使得KLEE能够精确的处理类型不安全的内存访问。KLEE要求数据对象有具体的大小，所以它不能支持大小不确定数据。KLEE不能直接支持指针，它处理指针的方法是通过一种插桩方法确定指针指向的对象。对于指针解引用，首先通过一个记录变量到其字节数组的映射表找到指针指向的对象对应的字节数组，然后计算指针相对这个字节数组的偏移量，最后把偏移量作为下标访问数组的元素。路径条件是关于字节数组的约束，对数组进行推理是STP的关键。STP通过对数组推理进行优化保证了它的性能。

### 2.2.2 LLVM介绍

由于KLEE基于LLVM，所以对LLVM做一个介绍。LLVM项目是模块化的、可重用的编译器和工具链技术的集合[24]。可以说是一个编译器框架，基于LLVM编写的编译器有LLVM-GCC和CLANG等。Clang 是一个 C++ 编写、基于 LLVM、发布于 LLVM BSD 许可证下的 C/C++/Objective C/Objective C++ 编译器。

LLVM命名最早源自于底层虚拟机（Low Level Virtual Machine）的缩写，由于命名带来的混乱，现在LLVM官方主页上已经声明和虚拟机没什么关系，LLVM就这个项目的全称。

LLVM利用其代码表示和编译器框架设计，以对程序员透明的方式，支持任意程序的全程（lifelong）分析和转换。LLVM的 全程代码优化包括：连接时执行的过程间优化、软件安装时的目标机器相关优化、 运行时执行的动态优化以及在程序不运行的空闲时间执行的剖面导向的（profile-guided）优化。全程分析和转换不仅可以用来优化程序，还可以用来做过程间静态分析，比如静态调试、内存泄漏检测等。另外，全程分析和转换也可以用来做可靠安全性质检查等。

LLVM的全程分析和转换基于LLVM的两个关键部分：代码表示和LLVM编译器框架。LLVM编译器框架利用代码表示提供了支持全程分析和转换的五种能力：

* 一致的程序信息：LLVM编译器在软件整个生存期一直保存着LLVM汇编 代码，这使得可以在任何时期对代码进行分析和优化。
* 编译型代码生成：对于性能关键的程序，可以把程序直接编译为本地机器代码。通过使用复杂的代码生成技术，可以保证生成的本地机器代码的高效性。
* 基于用户的剖面（profiling）和优化：LLVM在运行时向用户收集剖面信息，并把这些信息应用到基于剖面的转换中。这样，软件就能反映用户的 具体使用习惯。
* 透明的运行时模型：LLVM不指定对象模型、异常语义或者运行时环境， 这使得LLVM能编译支持任何语言的程序的分析和转换。
* 统一的全程序优化：语言独立性使得LLVM可以以统一的方式编译和优化 程序的所有代码，包括系统库和语言特定的运行时库。

LLVM的代码表示的特点是，提供支持复杂分析和转换的程序高级信息，比如类型，同时又可以表示任意程序，并支持各种优化。LLVM的指令集抽象了一般处 理器的关键操作，同时避免了机器特定的部分，比如物理寄存器、流水线等。 LLVM指令集比较简单，只有31个操作码，多数指令是三地址码的形式。LLVM 包含无数多个有类型的虚拟寄存器，这些寄存器可以保存基本类型的数据（布尔、 整数、浮点数和指针）。寄存器在LLVM汇编代码中是SSA（Static Single Assignment） 形式。内存位置不是SSA形式，原因是，一个指针可能指向多个内存位置。LLVM 汇编代码中，函数的CFG（Control Flow Graph）已经被构造出来，这一点十分有 用，因为很多静态分析依赖于函数的CFG，包括符号执行。这里CFG的节点是基本 块（basic block），而不是单个语句。LLVM汇编代码中的数据有确定的类型，它 提供一个语言独立的类型系统，包括四个简单类型：void、布尔、整数和浮点数， 以及四个导出类型：指针、数组、结构和函数。这个简单的类型系统可以实现绝 大多数高级语言的类型，比如，C++中的类可以使用结构、函数以及函数指针的数 组的组合实现。指令getelementptr用来实现指针算术，它可以计算组合类型（结 构或者数组）数据的成员的地址。getelementptr使得在LLVM汇编代码中附带类型信息成为可能。

在LLVM的汇编代码中，所有的堆数据使用malloc指令分配内存，返回有类型 指针。所有栈中的数据使用alloca指令分配内存，并返回有类型指针。全局变量和 函数定义定义了一个执行相应对象的地址。这样，所有的可寻址对象都有指向它 的指针，所有的内存操作，包括函数调用，都是通过指针进行执行。在内存和 寄存器之间进行数据传递是通过load和store指令来做的。LLVM的代码表示中的另 一个创新点是使用invoke和unwind两个指令实现了对高级语言中的异常处理的支 持。这种机制支持一种抽象异常处理模型，机器独立。

### 2.2.3 内存模型

内存模型的作用是跟踪符号状态。为了支持一些复杂的语法成分，比如类型不安全的内存访问（比如，C代码中常常出现的类型转换），内存模型需要足够精确。内存模型处理数据的方式与数据的类型和生存期密切相关。与其他静态分析一样，符号执行也需要面对指针和别名问题。同时，符号执行必须处理一些特殊问题，比如函数的参数是包含无界数据的复杂数据结构。当程序规模较大时，需要记录大量的符号状态，数据读写的时间消耗也不可忽视。这时，内存模型的性能（可扩展性）成为制约其实用性的关键。鉴于这些问题，设计一种精确、具有一定的可扩展性（calability）的内存模型，支持各种数据类型和语法成分，是推动符号执行深入、广泛应用的关键。

底层系统软件包括软件开发中使用的编译器、设备驱动程序、操作系统等， 它们是其他软件的基础，在整个软件系统中发挥着重要的作用。系统软件对正确 性有着更高的要求，因此，它们是形式化验证的主要目标。系统代码的一个特点 是代码中经常出现类型不安全的内存访问（type-unsafe memory access）。类型不 安全的内存访问主要包括类型转换和指针算术。比如，把一个数据结构转换为字 节数组。指针类型转换是一个典型情况。系统代码常常把内存看作无类型的字节 序列，通过指针类型转换以不同的方式读写一个内存位置。指针算术通过一个指 针与整数之和计算临近的数据对象的地址。比如，C语言规定的指针算术针对以指 针方式引用数组元素的场景，要求运算前的指针以及运算结果产生的指针指向数 组内的元素或者数组最后一个元素之后的一个对象。

为了能够分析和验证底层系统软件，内存模型必须能够精确描述类型不安全 的内存访问所带来的影响。类型不安全的内存访问的背后原因是系统代码把内存 视为无类型的字节数组，因此内存块在字节粒度可区分是精确的内存模型的关键。

一个极端是把内存空间视为单一的一个字节数组。很明显，这种内存模 型是完全精确的，它能够精确的表示所有的内存访问。但是，它的可扩展性很差。 随着硬件价格的降低，现代计算机系统的可用内存很大（一个普通微机的内存往 往是2G），程序的内存消耗也十分巨大。因此，精确的表示每一个内存位置的分 配和访问代价很高。字节数组内存模型对即使很小的代码段也很难胜任。另一 些内存模型则假设代码没有动态内存分配并且是类型安全的，这种内存模型可 以成功处理数十万行代码，但是不能精确分析系统软件。

（i）单体内存模型

单体内存模型（monolithic memory model）在系统软件的静态分析中应用广 泛，KLEE的内存模型本质上就是单体内存模型。单体内存模型的基本思想是把内存空间划分为不相交的区域（region），通过相应对象的程序符号引用识别区域。区域是具有相应数据对象大小的字节数组。指针表示为一个二元组，包括对象引用和字节偏移量，对象引用表示指针指向的区域，字节偏移量是指针指向的位置相对于区域开始处的字节偏移量。单体内存模型有三种数据类型：整数int、对象引用ref以及指针ptr=ref\*int。单体内存模型的语义由三个映射组成。

* Mem: ptr->{ptr,int}
* Alloc: ref->{allocated, unallocated}
* Size: ref->int

（ii）Burstall内存模型

Burstall内存模型是在类型安全代码的分析中广泛使用的内存模型。它的基本思想是不但把内存划分为不相交的区域，而且使用数据的类型区分不同的内存位置。Burstall内存模型的语义如下所示。

* Mem:ptr→value
* Alloc:ref→{allocated，unallocated}
* Size:ref→int
* Type:ptr→type

其中，type类型定义一些类型常量， 类型常量表示某一个内存位置的类型。比如一个整数指针对应的内存位置的类型 值为$intP。映射Men表示内存位置存储的值，通过指针和类型对标识内存位置。 映射Type表示一个内存位置的类型。Alloc和Size同单片内存模型。Burstall内存模 型中的类型信息使得它具有较好的可扩展性，但是，它不够精确，不能精确分析 包含类型不安全的内存访问的系统软件。

（iii）数据类型

数据类型定义了一组数据对象以及创建和操纵它们的操作集合。一个数据 类型的基本组成部分包括：

* 属性：区别于其他数据类型的性质，比如数组的维数等；
* 值：这个类型的数据对象的所有可能的取值；
* 操作：作用于这个类型的数据对象的操作，比如以下标访问数组元素等。编程语言实现数据类型需要定义相应数据对象的存储和操作的实现，包括在程序执行过程中相应数据对象在计算机存储器中的存储表示和根据特定于存储表示的算法或过程定义操作的实现。

常见的数据类型有标量数据类型、指针、结构化数据类型和抽象的数据类型等。标量数据类型包括枚举类型、字符型、数字类型和布尔类型。结构化数据对象由其他对象集合而成，这些对象称为成员。典型的结构化数据类型有数组、结构等。抽象的数据类型有面向对象编程中的类等。

（iv）数据的生存期

数据的生存期（lifetime）是指程序运行时为数据对象分配存储空间的时期。数据的生存期分三种：静态生存期、局部生存期和动态生存期。静态生存期指生存期从程序开始执行一直持续到程序结束。在C程序中，所有的外部变量以及定义中有static关键字的局部变量都有静态生存期。局部生存期指生存期从块或者函数的进入持续到块或者函数的退出。形式参数、多数局部变量都有局部生存期，这样的局部变量被称为自动变量。动态生存期指生存期由程序员控制。拥有动态生存期的动态数据对象往往通过特定的库函数，比如C中的malloc等产生。

（v）内存模型功能

符号执行系统的解释器执行代码的符号语义，根 据计算结果更新内存模型中的符号状态，并在分支语句处调用约束求解器判定约束公式。约束求解器是针对某种理论（比如一阶逻辑公式形式的包含等词的整数线性算术）的定理证明器，作用是判定约束公式的可满足性。内存模型的主要作用是跟踪符号状态。内存模型的功能包括以下几个方面：

* 跟踪符号状态；
* 定义内存操作的语义；
* 处理内存模型特定的约束公式，以方便约束公式的判定。

根据前面的定义，符号状态是一个三元组，IP跟踪程序的执行， 关于PC要解决的工作是是化简和求解，这是约束求解器的职责。M是变量到其符号值 的映射，是内存模型的基础和关键部分。M需要直接或间接地实现两个映射：Φ 和Ψ，如图3.2所示。Φ表示为可寻址对象分配的内存位置，Ψ表示内存位置上存储的符号值。其中，Denotation表示可寻址对象的程序符号的集合，Location表示内存位置的集合，SymbolicValue表示符号表达式的集合。

要在静态分析中使用一个内存模型，必须为编程语言定义针对这个内存模型 的内存操作语义。具体包括内存模型直接支持的数据类型（往往是编程语言的数 据类型的一个很小的子集）、如何进行数据对象的读写操作、如何建模动态内存 分配（比如C语言中的malloc()函数），以及内存操作相关的其他语法成分的语义。 比如，单体内存模型定义的动态内存分配的语义。

另外，内存模型需要处理内存模型特定的约束公式，以方便约束公式的判定。 路径条件来自路径上的分支条件，通过把变量替换为相应的表达式，从分支条件 直接得到的路径条件与内存模型是密切相关的。符号执行系统可把内存模型 和约束求解器紧密耦合在一起，也可把他们分离开来，由内存模型负责路径条 件的预处理。这样，经过预处理的约束公式是某种标准形式，就可以使用通用的约束求解器来判定。单体内存模型（monolithic memory model）把指针表示为一个二元组，则两个指针相等的谓词就变成了两个二元组的相等，不能处理向量的约束求解器 就不能判定这样的约束公式。KLEE把内存空间视为多个字节数组，因此，它使用的约束求解器STP必须能够高效的求解数组约束。

（vi）指针和别名

指针分析是指静态地确定指针的运行时取值，别名分析是指判断同一存储位置的多种访问方式。从指针分析的结果中可以得到别名信息，同样，从别名分析的结果中也可以得到一些指针信息。

指针分析算法根据其目的不同可以分为两种。第一种的目的是处理一般的、比较简单的指针。这些算法是域不敏感的，也就是不区分结构化类型数据中的不同成员——例如结构中的域。此类别的一些早期的算法还限制指针必须为单级指 针。第二种算法则集中于处理递归结构，例如链表、树等，出现了许多处理C语言程序的此类算法。别名分析算法可以划分为不同的精确度，精确度的标准有流不敏感算法、流敏感算法、上下文不敏感算法、上下文敏感算法、路径不敏感算法和路径敏感算法等。其中，流不敏感算法的精确性最低，而路径敏感算法的精确度最高。一般而言，随着精确度的提高，算法的计算代价随之升高，可扩展性则随之降低。流不敏感别名分析算法为整个程序只生成一个别名分析结果，即内存位置之间的指向图，用以表示所有可能的别名关系。比较著名的流不敏感算法的基本思路是为每个指针表达式分配一个类型，代表该表达式所有可能指向的内存位置。为表达式分 配类型的过程就是类型推理（type inferencing）过程。对指针表达式赋值语句的处理方法是，为该语句左端表达式的类型和右端表达式的类型生成一个约束条件，其含义是语句执行之后，左端表达式与右端表达式指向同一个内存位置。通过求解约束，即可得到别名分析结果。

流敏感的别名分析算法可以使用传统的数据流分析框架来表示。在每个程序 点，可以计算出多种形式的数据流事实。在处理指针赋值语句时，通常先计算出 该语句消除了哪些别名信息，然后再计算出它生成的新的别名信息。在汇合结点 处，做别名信息的合并操作。上下文敏感指针分析算法同上下文不敏感算法相比，需要消除别名信息在过程间不可行路径上的传播。主要有两种方法。第一种方法为别名信息标记上下文信息，上下文标记用来区分在不同调用上下文中生成的别名信息。第二种方法是，在不同的上下文环境下，对过程的每一个调用，采用宏扩展方式进行相互独立的分析。这样，一个过程可能会被分析多次，导致了性能代价。

指针和别名问题也是符号执行必须面对和解决的问题。指针和别名分析算法 代价较高，尤其是精确性高的算法。多数符号执行系统不支持指针，出于性能方 面的考虑，往往使用十分简单的算法（比如流不敏感算法）处理指针和别名问题。使用流不敏感的指针和别名分析结果做路径敏感的符号执行，似乎存在悖论。虽然通过使用各种近似方法可以得出有意义的结果，但这确实严重影响了符号执行的精确性。

指针和别名分析只关心数据之间的指向关系，而忽略变量的具体值和内存地 址，而符号执行本身就需要精确跟踪内存状态，包括变量的值。所以，符号执 行可以使用更精确的方法处理指针和别名问题，而不仅仅是使用现有的指针和别 名分析算法。

（vii）大小不确定的数据

这里的大小不确定数据是指无界复杂数据和大小是符号值的动态分配的数据。无界复杂数据包括链表（list）、树（tree）等，是Java、C++等现代编程语言的一个高级结构，在代码中经常出现。符号执行必须面对以这些数据作为输入的函数所造成的困难。惰性初始化算法可以处理无界复杂数据，但是惰性初始化算法是一种数据初始化方法，符号执行中的内存模型必须为无界复杂数据和相应的处理算法提供支持。

一些编程语言（C、C++等）的动态内存分配机制为灵活地进行程序设计提供了方便。动态内存的分配与使用是在程序代码中按照算法的要求，由程序开发人员控制的。但是，动态内存分配却给符号执行造成了困难。考虑这么一个C语句：int\*p=malloc(n\*sizeof(int))，其中n是依赖输入的符号值。这个语句动态分配的n元整数数组的大小就是不确定的。内存模型必须能够支持大小是符号值的动态分配的数据。

为了建模无界复杂数据和大小是符号值的动态分配的数据，内存模型必须允许这些数据可能的动态收缩或扩展，而不能在这些数据生成时对其大小做任何假设。把内存模型视为一个字节数组的方法不支持大小不确定数据，原因是如果数据的大小不能确定，就不知道它需要的数组元素的个数，进而就不能确定下一个数据在数组中的位置（对于大小是符号值的数据，会使得接下来的数据的索引都是符号值）。单体内存模型也不支持大小不确定数据，因为它的每一个区域都要求有确定的大小。

3 源程序分析工具设计与实现

## 3.1 需求分析

本课题研究的目的是设计并实现一个源程序分析工具，主要用于求出源程序中全局变量每处调用的位置、读写类型等信息。

全局变量每处调用需要获取的主要信息有：

* 变量名
* 行号
* 读或者写
* 全局变量调用所在函数名

常见的全局变量类型：

* 基本类型：

C语言中定义的基本类型，int，char，double等。

* 指针：

包括指向基本类型的指针，和其他一些更复杂的类型的指针，如：数组，指针，结构体等。

* 数组：

包括基本类型的数组，指针数组，结构体数组，多维数组等。

* 结构体
* 动态分配的空间

程序中使用malloc等从堆区动态分配的空间。

## 3.2全局变量在中间文件中的表示

对待分析的源程序，首先使用llvm-gcc或clang编译成中间文件，然后再进行分析。在编译时，如果加上-S参数，编译成的中间文件后缀名是.s，是以纯文本方式存储的，可以直接阅读文件内容；而加上-c参数，编译成的中间文件后缀名为.o，以二进制方式存储，不能阅读，但内容与.s文件是相同的。下面对一些常见的全局变量在中间文件中的表示做一个分析。

1．基本类型

以int型为例，变量iA是在函数体外声明的全局变量，图3.1是变量iA的声明和赋值语句在中间文件中的表示。其中!dbg !25是这条语句调试信息的标识，在中间文件的最后，有对应的调试信息（编译时加上-g参数，生成的中间文件中会存在调试）。可以看到，全局变量iA在中间文件中，以@iA命名，@是全局变量的标识。中间文件中，用i+数字表示位数，int型在这里是4字节、32位，因此用i32作为前缀。赋值语句在中间文件中是一条store语句，将32位的常量1存入iA对应的内存单元中。

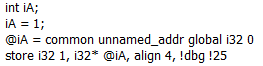


图3.1 int型全局变量

2．指针

如图3.2所示，是一个指向int型变量的全局指针在中间文件中的表示。

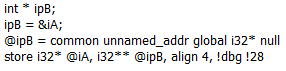


图3.2 指向int的全局指针

3．数组

如图3.3所示，是一个int型全局数组在中间文件中的表示。

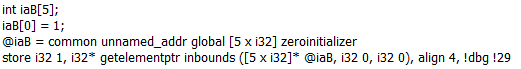


图3.3 int型全局数组

4．结构体

如图3.4所示，是一个结构体变量的声明和调用，以及在中间文件中的表示。

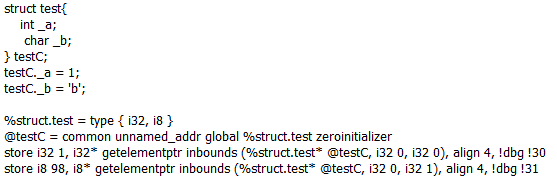


图3.4 自定义结构体全局变量

5．动态分配的空间

如图3.5所示，ipB是一个int型的全局指针，ipC是在函数内声明的int型的局部指针，通过malloc分配4个字节的空间，并进行赋值操作。

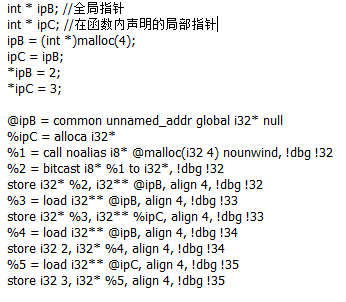


图3.5 动态分配的空间

可以看出，对于在函数体外声明或定义的变量，在中间文件中会以@+变量名方式命名。所以，可以通过识别@判断一个变量是否是全局变量。但是，对于动态分配的空间，从实际意义上来说应该认为是全局变量，但无法用这种方法判断的。

## 3.3 具体实现

用来保存全局变量调用信息的结构如下：

struct VarCall{

int line；

int assemblyLine；

int writeOrRead；

std::string name；

std::string functionName；

}

其中assemblyLine是中间文件中的行号。对于a++这种类型的语句，先读变量a的值到中间变量，然后将中间变量的加一，再写入变量a中。实际上在中间文件中是三个操作。由于writeOrRead指的是源文件中变量是读还是写，对于上面的情况而言，既进行了读，也进行了写，因此单独用2来表示（0表示读，1表示写）。

用变量类来处理变量的信息，为每一个变量生成一个该类的实例。对于同一个变量的多处调用信息，用一个vector来进行管理，这个vector作为变量类的类成员。用一个单独的类管理所有变量，主要负责维护一个map，为变量名和对应的对象做映射，方便查找变量和插入新的调用信息。

整体流程是，使用KLEE对中间文件进行模拟执行，当执行到变量读写语句时，判断该变量是否是全局变量。若是全局变量，则获取它的行号、中间文件行号、读或写、变量名和所在函数名等信息，填充struct，并插入到相应变量的调用信息列表中。

## 3.3 测试分析

对一个简单的多线程程序进行测试，程序在附录中，测试结果如表3.1所示。

表3.1 测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 读写 | 函数名 | 变量名 |
| 22 | 1 | main | semid |
| 24 | 0 | main | semid |
| 26 | 0 | main | semid |
| 61 | 0 | subp1 | semid |
| 63 | 0 | subp1 | semid |
| 74 | 0 | subp2 | semid |
| 29 | 0 | main | p1 |
| 30 | 0 | main | p2 |
| 62 | 0 | subp1 | a |
| 75 | 1 | subp2 | a |

# 4 总结与展望

本文首先介绍了程序分析技术的发展历史，介绍了符号执行和动态符号执行技术的概念。然后重点阐述了符号执行技术的基本原理，和面临的问题，并用一个示例演示符号执行的过程。接着，对动态符号执行工具KLEE及LLVM进行了说明。

在本文中，对各种类型的全局变量在LLVM编译的中间文件里的表示进行了探讨。设计并实现了一个基于KLEE的源程序分析工具，主要分析源程序中的全局变量调用信息。通过一个基本的多线程程序对分析工具进行了测试。可以说，这个工具本身还是有很多不足之处的。对于一些复杂的情况，比如局部指针指向的动态分配空间，无法正确地识别。

目前，本课题的研究成果，是对源程序中的全局变量做分析，这只是源程序分析的一部分。在目前的研究成果上，还可以进一步地做数据竞争检测，内存泄露检测等分析。

# 致 谢

大学四年转瞬即逝，在毕业前的最后这段时间里，时常想起这四年里，和老师、同学和朋友在一起的欢乐时光。感谢四年中，所有教导过我，帮助过我的老师。感谢计算机科学与技术1002班的同学们，因为有大家的陪伴，大学生活才这么精彩。

首先要感谢我的导师李国徽教授对我的教导和帮助。虽然和李老师接触的时间不算长，但您的睿智和博学还是让我很佩服。感谢万齐智学长和汪雄峰学长在毕业设计过程中对我的无私帮助。

感谢我的家人和朋友对我的关心、支持和鼓励。

最后，要感谢我的母校，四年的大学生活，我在校园里留下了很多美好的回忆。

# 参考文献

[1] Ding Y．SSL implementieren–aber sicher![J]．2014．

[2] 林锦滨，张晓菲，刘晖．符号执行技术研究[J]．全国计算机安全学术交流会论文集，2009：404-408．

[3] Godefroid P，Klarlund N，Sen K．DART：directed automated random testing[C]．ACM Sigplan Notices．ACM，2005，40(6)：213-223．

[4] 文昌辞，王昭顺．软件测试自动化静态分析研究[J]．计算机工程与设计，2005，26(4)：987-989．

[5] 林阿龙．C 源程序分析及理解的辅助工具—CAAT[J]．小型微型计算机系统，1992，1：001：9-15．

[6] URL：

[http：//www.ibm.com/developerworks/cn/rational/products/purifyplus](http://www.ibm.com/developerworks/cn/rational/products/purifyplus)

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

[8] 杨宇，张健．程序静态分析技术与工具[J]．计算机科学，2004，31(2)：171-174．

[9] 赵跃华，阚俊杰．基于符号执行的测试数据生成方法的研究与设计[J]．计算机应用与软件，2014，31(2)：303-306．

[10] Zamfir C，Candea G．Execution synthesis：a technique for automated software debugging[C]．Proceedings of the 5th European conference on Computer systems．ACM，2010：321-334．

[11] Godefroid P．Compositional dynamic test generation[C]．Acm Sigplan Notices．ACM，2007，42(1)：47-54．

[12] Godefroid P，Levin M Y，Molnar D A．Automated Whitebox Fuzz Testing[C]．NDSS．2008，8：151-166．

[13] Sen K，Marinov D，Agha G．CUTE：a concolic unit testing engine for C[M]．ACM，2005．

[14] Burnim J，Sen K．Heuristics for scalable dynamic test generation[C]．

Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering．IEEE Computer Society， 2008：443-446．

[15] Sen K，Agha G．CUTE and jCUTE：Concolic unit testing and explicit path model-checking tools[C]．Computer Aided Verification．Springer Berlin Heidelberg， 2006：419-423．

[16] Cadar C，Ganesh V，Pawlowski P M，et al．EXE：automatically generating inputs of death[J]．ACM Transactions on Information and System Security (TISSEC)，2008，12(2)：10．

[17] 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．

[18] Boonstoppel P，Cadar C，Engler D．RWset：Attacking path explosion in constraint-based test generation[M]．Tools and Algorithms for the Construction and Analysis of Systems．Springer Berlin Heidelberg，2008：351-366．

[19] Visser W，Havelund K，Brat G，et al. Model checking programs[J]. Automated Software Engineering，2003，10(2)：203-232．

[20] Jayaraman K，Harvison D，Ganesh V，et al．jFuzz：A Concolic Whitebox Fuzzer for Java[C]．NASA Formal Methods．2009：121-125．

[21] Tillmann N，De Halleux J．Pex–white box test generation for．net[M]．Tests and Proofs．Springer Berlin Heidelberg，2008： 134-153．

[22] Visser W，Pǎsǎreanu C S，Khurshid S．Test input generation with Java PathFinder[J]．ACM SIGSOFT Software Engineering Notes，2004， 29(4)：97-107．

[23] Cadar C，Ganesh V，Pawlowski P M，et al．EXE：automatically generating inputs of death[J]．ACM Transactions on Information and System Security (TISSEC)，2008，12(2)：10．

[24] URL：<http://llvm.org/>

# 附录 测试程序

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <sys/types.h>

#include <linux/sem.h>

#define LOOPS 10

int semid;

int a=0;

pthread\_t p1,p2;

void P(int semid, int index);

void V(int semid, int index);

void \*subp1();

void \*subp2();

int main()

{

union semun semops;

int res;

semid = semget(IPC\_PRIVATE, 2, IPC\_CREAT|0666);

semops.val = 0;

res = semctl(semid, 0, SETVAL, semops);

semops.val = 1;

res = semctl(semid, 0, SETVAL, semops);

pthread\_create(&p1, NULL, subp1, NULL);

pthread\_create(&p2, NULL, subp2, NULL);

pthread\_join(p1, NULL);

pthread\_join(p2, NULL);

semctl(semid,0,IPC\_RMID,0);

return 0;

}

void P(int semid, int index)

{

struct sembuf sem;

sem.sem\_num = index;

sem.sem\_op = -1;

sem.sem\_flg = 0;

semop(semid, &sem, 1);

return;

}

void V(int semid, int index)

{

struct sembuf sem;

sem.sem\_num = index;

sem.sem\_op = 1;

sem.sem\_flg = 0;

semop(semid, &sem, 1);

return;

}

void \*subp1()

{

int i;

for (i=0; i<LOOPS; i++)

{

P(semid, 0);

printf("subp1:a=%d\n", a);

V(semid, 1);

}

return;

}

void \*subp2()

{

int i;

for (i=0; i<LOOPS; i++)

{

P(semid, 1);

a++;

printf("subp2\n");

V(semid, 0);

}

return;

}