ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.10, October 2008, pp.2550−2561
DOI: 10.3724/SP.J.1001.2008.02550

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

C 代码 API 一致性检验中的等值分析*

孟 策[1,2+], 贺也平[1], 罗宇翔[1,2]

[1](中国科学院 软件研究所,北京 100190)

[2](中国科学院 研究生院,北京 100049)

Value Equality Analysis in C Program API Conformance Validation

MENG Ce[1,2+], HE Ye-Ping[1], LUO Yu-Xiang[1,2]

[1](Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: E-mail: mengce@ercist.iscas.ac.cn

Meng C, He YP, Luo YX. Value equality analysis in C program API conformance validation. *Journal of Software*, 2008,19(10):2550−2561. http://www.jos.org.cn/1000-9825/19/2550.htm

Abstract: In the attempt to apply static analysis method to API (application programming interface) conformance validation, It is noticed that the gap between numeric value based temporal properties in API specification and variable symbol based predicates extracted through static analysis. Therefore, the paper first investigates value equality relations between variable symbols in C program, and then designs an ECS (equality class space) based value equality analysis method. This flow-sensitive method can maintain the correspondence between variable symbol domain and numeric value domain during the process of API conformance validation, and in addition, can effectively support the optimization of subsequent analysis by hiding all irrelevant information except value equality relations.

Key words: static analysis; API conformance validation; equality analysis; points-to analysis

摘 要: 在实际中对C代码进行API一致性检验的过程中发现,API(application programming interface)规范大都涉及以数值为论域的时序性质,与在静态分析过程中所能获取的以变量符号为占位符的独立语义之间存在分析上的缺口.在仔细考察C代码变量符号间等值关系的基础上,给出基于值等价类空间的等值分析方法.这种流相关的分析方法不仅可以在 API 一致性检验的过程中维护变量符号域和数值域之间的对应关系,而且由于能够屏蔽等值关系以外的其他信息,还可以为后继分析的优化提供有力的支持.

关键词: 静态分析; API一致性检验;等值分析;指针分析

中图法分类号: TP311 **文献标识码**: A

随着硬件性能的不断提升,计算机被赋予越来越艰巨的任务,软件作为沟通人类思维和计算机硬件的桥梁,其重要性不言而喻.尽管对软件设计开发方法的研究从未停止过,自顶向下逐步求精、面向对象等理念也已经在实际的项目中得到充分的体现,但是,软件设计开发效率的提升依旧缓慢,难以跟上硬件的发展速度.这导致

* Received 2007-08-19; Accepted 2007-10-12

开发人员面对的任务规模越来越大、逻辑越来越复杂,即使整个设计开发过程都严格遵从保障规程,在软件系统中仍然难免由于疏漏留下隐患.为了应对这一问题,人们在软件保障中引入测试方法并对其进行了系统的研究.合理地设计测试方案、选择测试用例可以有效地识别软件实现与设计的不一致,定位潜藏的问题.但是,测试作为一种动态分析方法无法确保对软件系统所有执行路径的覆盖,没有得到检验的路径中存在的问题将被遗漏.静态分析方法由于恰好可以弥补这一不足而开始在软件保障领域中崭露头角.随着代码静态分析技术应用于软件保障的研究日益深入,其涉及的问题越来越复杂,语义规模与代码间的距离也被逐步拉大.这导致分析需要提取的语义在代码中的表现形式千变万化,难以找到简明的模式准确地加以描述,传统的单纯基于词法匹配的方法渐渐变得不再适用.为此,人们开始将问题拆分,希望通过细化语义粒度降低其提取的难度.举例来说,C 代码中造成内存泄露的行为可以用申请堆内存空间和释放堆内存空间这两个相对独立的语义加以描述,它们都可以比较准确地得到识别.

API 规范用于指明软件系统各个功能模块之间交互所需遵循的准则,是开发人员思维交流的边界,因此也是易于出现问题的位置,对于 API 功能、关注点理解的偏差都可能在软件系统中引入缺陷.API 一致性检验正是用于识别这种有悖于 API 使用规则的行为,由于涉及的独立语义具有简明而统一的表现形式——API 调用,因此十分适于依据上述拆分语义的思路基于静态分析进行.此前,静态分析方法在软件保障中的成功应用,如 Chen 等人使用 MOPS(modelchecking programs for security properties)[1]验证 EROS(the extremely reliable operating system)[2]内核的安全不变量[3];Zhang 等人结合 CQUAL 和动态分析方法检验 LSM(Linux security modules)的授权点设置[4,5],都基于对特定 API 使用情况的考察.在从代码中提取出涉及 API 调用的独立语义后,如何确定这些语义之间的关系就成为首先需要解决的问题.由于代码是面向计算机硬件的,API 规范涉及的时序性质是以数值为论域的,与静态分析能够提取的以变量符号为占位符的语义之间存在分析上的缺口,因此经常需要判断处于不同执行文境中的变量符号是否具有相同的取值.特别是当涉及跨函数的分析时,实参、形参间的隐式传值操作更加凸现这一需求.通过对 C 代码中变量符号间等值关系的研究,我们设计了基于值等价类空间的等值分析方法,后继分析可以使用值等价类替代变量符号携带独立语义,达到联系相关独立语义的目的.这种流相关的分析方法不仅可以在 API 一致性检验的过程中维护变量符号域和数值域之间的对应关系,而且由于能够屏蔽等值关系以外的其他信息,还可以为后继分析的优化提供有效的支持.

本文主要讨论 C 代码 API 一致性分析中的等值分析.第 1 节简要介绍相关研究.第 2 节首先讨论等值分析的必要性,之后引入符号间等值关系和值等价类空间的概念,最后给出基于值等价类空间的等值分析方法的大体思路.第 3 节讨论该等值分析方法针对 C 代码的实现.为了检验该方法,我们实际考察了 FreeBSD 6.0 内核中锁机制的使用情况,文中第 4 节对此实验进行简要的介绍.最后给出结论并对未来的研究工作进行展望.

1 相关研究

将代码静态分析方法应用于软件保障的最初尝试基于词法分析,通过模式匹配直接从代码中提取分析所关注的语义.随着研究的深入,人们希望处理的问题越来越复杂,为了确保语义提取得准确,人们开始讨论如何拆分问题以达到细化语义粒度的目的.MOPS[1]和 MC(meta-level compilation)[5]是这方面研究的先驱,它们都使用有限自动机模型描述代码包含的时序性质.MOPS 基于变量符号传递独立语义,由于它不进行跨函数的分析,而函数内传值操作的情形又十分有限,因此具有很好的实用效果,Shapiro 曾邀请 Chen 使用 MOPS 通过考察特定 API 的一致性对 EROS 内核的安全不变量进行验证[2].MC 衍生出的产品 Coverity 是当今业界的旗帜,曾实用于 Linux 内核的全面缺陷检测.

微软长期支持的 SLAM[6]首先将 C 代码抽象为布尔语言代码,进而从中寻找存在问题的执行路径.与 MOPS 和 MC 相比,SLAM 引入了针对分支条件的布尔值分析,能够剔除分析结果中的不可达路径,其衍生出的 SDV(static driver verifier)专用于 Windows 驱动开发 API 的一致性检验.与 SLAM 有很深渊源的 BLAST(Berkeley lazy abstraction software verification tool)[7]针对布尔语言提出惰性抽象[8]的概念,提高了使用该方法进行分析时的效率,曾用于 Windows 和 Linux 下驱动程序的检验[9].

在静态分析的过程中还可以利用注释辅助语义的提取,其思路是由开发人员保证部分代码的语义并通过注释指明.使用该方法可以使提取出的语义包含更复杂的逻辑,从而降低后继分析的难度.以 Splint[10]为例,如果能够合理使用其提供的注释机制,可以有效地减少分析时出现的误报和漏报.而 CQUAL 基于类型修饰符理论[11,12],通过将关注的问题转化为修饰符运算格检测插入代码关键位置的类型修饰符之间的冲突.插入代码的类型修饰符将完整的执行路径进行了切分,从而削减了跨函数分析的深度.Zhang 等人曾使用 CQUAL 检验 LSM 授权 API 的一致性[3].

此外,API 的使用规则还可以通过类型状态规范加以描述,与有限自动机模型类似,类型状态规范有能力为大多数库和接口的使用规则建模[13].近年来,许多研究都尝试对命令式语言代码的类型状态性质进行验证或检查,比如 ESP(error detection via scalable program analysis)[14].为了跟踪软件系统对特定抽象概念的操作,这些研究大都引入别名分析,将验证过程划分为两个阶段:首先进行指针分析(points-to analysis)对数据流进行跟踪,之后以此为基础完成对类型状态的考察[14,15].

目前,国内对于静态分析的研究刚刚起步,工作大都围绕总结[16,17]和针对特定问题的静态分析方法展开,比如针对缓冲区溢出静态分析中流不相关的指针分析方法的研究[18]等.

2 等值关系和基于值等价类空间的等值分析

2.1 等值分析的必要性

如前文所述,目前代码静态分析大都遵循拆分语义的思路将语义提取和语义分析分开处理.在这种情况下,如何将相关的独立语义联系起来就成为语义分析首要解决的问题.目前较常见的思路是用变量符号作独立语义传递的载体,但由于 API 规范所涉及的时序性质以数值而并非变量符号为论域,在依据变量符号关联独立语义时就经常需要判断处于不同执行文境中的符号是否具有相同的取值.

举例来说,库 Libiberty 中的 API asprintf 在头文件 Libiberty.h 中的声明如下所示:

```
/* Like sprintf but provides a pointer to malloc'd storage, which must be freed by the caller. */
extern int asprintf (char**,const char*,...) ATTRIBUTE_PRINTF_2;*
```

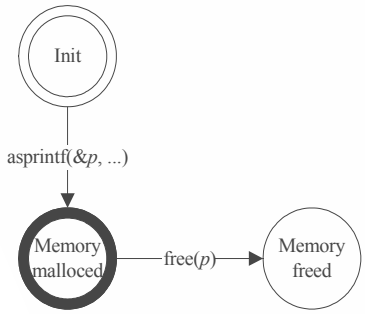

Fig.1 Finite automaton describing the usage rules of asprintf

图 1 描述 asprintf 使用规则的有限自动机模型

根据注释可知,在调用 asprintf 后,调用者需要显式地释放通过第一个参数返回的存储空间,否则会导致内存泄漏.对于这一规则我们可以建立如图 1 所示的有限自动机模型,将问题拆分为两个独立语义"调用 asprintf"和"释放堆内存空间",它们在 C 代码中都有简明的模式——包含 char*类型的占位符 *p* 的代码 asprintf(&*p*,…)和 free(*p*).这两种独立语义会引发对应不同堆内存空间的有限自动机实例的状态转移,如果有实例最终停留在"Memory malloced"状态,说明存在内存泄漏的现象.

在静态分析的过程中,我们可以从特定的执行路径中提取出前述两种独立语义,构成语义序列.显见,并非语义序列中的所有语义都共享同一有限自动机实例,如图 1 所示,我们可以通过语义模式中的占位符 *p* 将相关的独立语义联系起来.但是模型所包含的逻辑以数值为论域,即占位符应当属于数值空间而非变量符号空间.同时,由于静态分析无法确知特定执行文境中变量符号的取值,因此只能提取出以变量符号为占位符的语

* GCC(the GNU compiler collection)不仅可以支持 c89,ansi,c99 等 C 语言标准,还对其进行了扩展.编程者可以通过关键字__attribute__向 GCC 提供语法构件的一些附加信息,辅助 GCC 对 C 代码的静态检查和优化.此处的宏 ATTRIBUTE_PRINTF_2 展开后为__attribute__((__format__ (__printf__,2,3)))__attribute__((__nonnull__(2))),前者用于指明 printf,scanf 风格的函数中格式化字符串的位置及其参数的起始位置,而后者表示作为输入的格式化字符串不能为 null.

义,语义分析在这里出现了一个从变量符号空间到数值空间的缺口.

如果代码中的传值操作数目有限,我们可以近似地认为变量符号和数值间存在一一对应的关系.例如图 2(a)中的代码,由于在 asprintf (*&buffer*)和 free(*buffer*)之间变量符号 *buffer* 没有涉及传值操作,因此,通过 *buffer* 将这两个独立语义联系起来可以得到正确的语义分析结果.相反地,如果对于图 2(b)、图 2(c)中包含传值操作的代码仍使用变量符号 *buffer* 传递独立语义将导致错误的分析结果,会分别产生漏报和误报.这是由于,图 2(b)的代码中尽管显式地释放了 *buffer* 指向的堆内存空间,但是此时 *buffer* 的取值已经改变,不再是此前调用函数 asprintf 时返回的堆内存空间地址;而图 2(c)的代码中由于执行语句 free(*equivalent*)时变量符号 *equivalent* 与 *buffer* 等值,*buffer* 指向的堆内存空间隐式地得到了释放.由于 C 代码中的函数调用过程包含实参和形参间的隐式传值操作,因此,当涉及跨函数的分析时,即使代码中显式传值的赋值操作数目有限,我们仍旧无法回避上述问题,必须找到可以在静态分析过程中判断处于不同执行文境中的变量符号是否具有相同取值的方法才行.

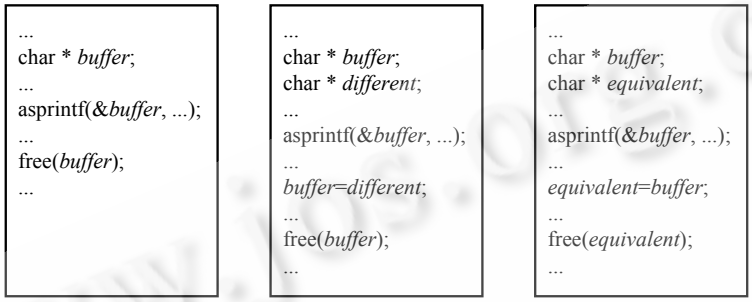

Fig.2 Illustrations on the usage of asprintf

图 2 调用 asprintf 的代码示例

2.2 等值关系和值等价类空间

在代码静态分析的过程中,我们需要确定以变量符号为占位符的语义之间的关系.由于代码是面向计算机硬件的,真正起到关联语义作用的是变量符号的取值,而这些取值往往是在软件运行时动态获取的,只能通过传值操作在不同的执行文境中传递,因此我们有必要关注这些操作引发的变量符号间的等值现象.

定义 1. 当两个变量符号由于传值操作具有相同取值时,我们称这两个变量符号间具有等值关系.

这种等值关系的建立依赖于传值操作,由于其他形式(如数值运算)引发的等值现象大都与语义分析所涉及的动态获取的变量符号取值无关,对于关联独立语义的助益不大,因此我们不予关注.传值操作除了可以显式地为其直接涉及的两个变量符号建立等值关系,还可能由于语言本身具有的特性,例如 C 语言中指针引起的静态别名现象,隐式地为其他变量符号建立等值关系.无论显式还是隐式,只要两个变量符号由于传值操作获得相同的取值,我们都认为它们建立了等值关系.同时,变量符号涉及的原有等值关系也会受到影响,可能会被打破.

显见,等值关系具有自反性、对称性和传递性,因此等值关系是一种等价关系.在软件任何一个特定的执行文境中,都可以依据等值关系将其中所有的变量符号划分到若干值等价类 VEC(value equality class)中.以图 3(a)中的代码为例,语句(3)执行后可以依据变量符号 $v1$,$v2$,$v3$ 的取值作如图 3(b)所示的划分.值等价类可以用其元素的取值来标识,显而易见,和数值之间具有一一对应的关系,如前例中值等价类$\{v1,v2\}$和 0 对应,值等价类$\{v3\}$和 3 对应.随着执行文境的推演,变量符号的取值可能发生变化,此时,它在值等价类空间中也会相应地从旧值对应的值等价类转移到新值对应的值等价类中.与变量符号不同,值等价类对应的数值不会在值等价类之间转移,它与值等价类之间的对应关系在软件的整个执行过程中都是稳定的.

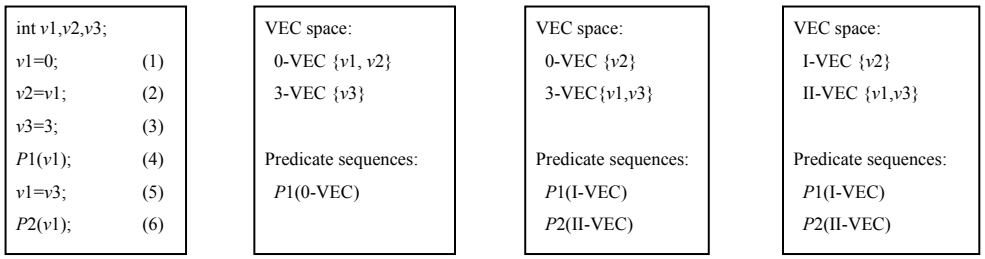

Fig.3 Illustrations on the correlation of related predicates with VEC

图 3 依据值等价类关联相关独立语义的示例

2.3 基于值等价类空间的等值分析

由于值等价类和数值之间存在稳定的一一对应的关系,我们可以将前述确定变量符号和数值之间对应关系的问题转化为确定变量符号和值等价类之间对应关系的问题.通过在分析过程中构建值等价类空间,可以将提取出的以变量符号为占位符的语义转化为以值等价类为占位符的语义,从而填补变量符号空间到数值空间的分析缺口.仍以图 3(a)中的代码为例,语句(3)执行后值等价类空间如图 3(b)所示.我们在语句(4)的位置上提取出以变量符号 *v*1 为占位符的语义 *P*1(*v*1),如前所述,将 *v*1 用其所属的 0 值等价类替换得到语义 *P*1(0-VEC).同理,我们在语句(6)的位置上提取出以变量符号 *v*1 为占位符的语义 *P*2(*v*1)时,值等价类空间如图 3(c)所示,因此,通过替换得到的语义不再以 0 值等价类为占位符,而是 *P*2(3-VEC).此时,分析提取的语义序列如图 3(c)所示,语义 *P*1 和 *P*2 用于占位符的值等价类不同,意味着在它们各自的执行文境中作为占位符的相同变量符号并不等值,因此不应将这两个独立语义联系起来.

由于在静态分析的过程中很难确定特定执行文境中变量符号的取值,因此,我们也无法在实际分析时如前例所示确知值等价类对应的数值.但我们真正关心的是变量符号间的等值关系,因此只需关注它们之间的传值操作就可以近似地还原值等价类空间,进而达到关联相关语义的目的.考虑图 3(a)中的代码,即使忽略语句(1)和语句(3),我们依旧可以得到如图 3(d)所示的分析结果,可以判断出语义 *P*1 和 *P*2 并不相关.当然,如果有变量符号通过赋确值间接地建立等值关系,上述近似就会导致偏差,但如前所述,由于我们在此需要关联的独立语义是 API 调用,而真正起到关联作用的数值往往都是在代码执行动态确定的,所以出现这一情形的可能性很小,其影响可以忽略.

2.4 与其他形式的等值分析方法的比较

等值分析如前文所述在代码静态分析中不可或缺,因此在相关研究中都或多或少地引入了类似的分析.比如前文提到的利用别名分析的方法,可以通过找出使用同一存储空间的变量符号有效地识别处于代码不同位置的同一抽象概念,进而将软件系统对该抽象概念的操作联系起来.但是这些指针分析方法都未能很好地结合后继的语义分析,由于缺乏针对性,往往为了平衡性能只提供流不相关的别名信息,这必然影响分析整体的准确性.而本文提出的等值分析方法是流相关的,可以在 API 一致性检验的过程中更加准确地维护变量符号域和数值域之间的对应关系.

目前流相关的等值分析方法大都基于符号值替换的思路.以图 2(c)中的代码为例,对于独立语义 free(*equivalent*)中的占位符 *equivalent*,通过回溯赋值语句 *equivalent=buffer* 将其替换为被视作数值的符号 *buffer*,从而能与之前涉及 *buffer* 的相关语义联系起来.与基于符号值替换的等值分析方法相比,构建和维护值等价类空间可以将等值分析从整个分析过程中独立出来,为后继分析提供简明而统一的访问界面.在语义提取的同时只需将代码中符号之间的传值操作提交给等值分析模块,就随时可以向该模块查询符号和值等价类之间的对应关系了.具体说来,引入值等价类空间具有以下两个方面的优势.其一是可以更加灵活地处理语言本身特

性对等值分析的影响.比如 C 代码中结构、联合、数组以及指针算术操作的存在使得指向特定存储空间的表示法千变万化,特别地,当这些特性引起的静态别名现象伴随传值操作出现时,动态别名现象会使符号值替换的过程变得十分繁琐,经常需要进行多次回溯.而基于值等价类空间的分析方法可以在接受传值操作时灵活地处理上述情形,将其统一转化为符号和值等价类之间的对应关系.其二是可以更有效地支持后继分析的优化.值等价类不由特定符号标识,屏蔽了与等值关系无关的信息,因此可以更加灵活地进行匹配,从而更有效地支持后继分析的优化.使用基于符号值替换的等值分析方法,等值的符号即使在相同上下文中携带同样的语义信息,也可能由于不同执行路径上传值操作的差异引起相关符号值的区别,最终导致后继分析的重复.而值等价类空间对后继分析屏蔽传值操作的细节,只提供符号间的等值关系.在上述情形中,尽管无法获知符号的确值,我们仍然可以依据一致的等值关系对分析关注的状态空间判定,避免无谓的重复分析.

3 针对 C 代码的等值分析实现

3.1 C代码中的传值操作

C 代码中,赋值语句所包含的显式传值操作和函数调用语句所包含的涉及实参、形参的隐式传值操作都会引起值等价类空间的改变.我们可以统一地使用二元组⟨*Exp*1,*Exp*2⟩描述这两种情形中表达式 *Exp*2 传值给 *Exp*1 的操作.我们可以在检验 API 一致性的同时从执行路径中提取出前述二元组的序列,利用其中包含的传值操作信息就能还原特定执行文境下的值等价类空间了.

如前文所述,我们的等值分析只是通过记录变量符号间的等值关系近似地还原值等价类空间,因此,并非所有的传值操作信息都需要保留,我们重点关注的是可能与 API 调用语义相关的、取值动态确定的变量符号.在 C 代码中的左值符号是有对应存储空间的数据,通常用于指明程序涉及的抽象概念并记录其状态,正好与上述关注的焦点契合.因此,我们选择传值操作中 *Exp*2 为“左值符号+整值”的指针算术操作(由于我们实现等值分析基于的 GCC4 中间代码已经对指针算术操作表达式进行了规范,在此可以不考虑它们在 C 代码中形式上的变化)作为重构值等价类空间的主要依据.对于 *Exp*2 不符合这一模式的传值操作,由于在无法确知 *Exp*2 取值时记录该操作的详细信息对于确定符号间的等值关系助益不大,在此可以忽略 *Exp*2 的内容将其统一地记为二元组⟨*Exp*1,*number*⟩.由于 C 语言中结构、联合、数组和指针算术操作的存在,对应相同存储空间的左值符号可能具有不同的表现形式,因此,在讨论还原值等价类空间的方法之前,我们需要先统一 C 代码中左值符号和指针算术操作表达式的表示方法.

3.2 左值符号的统一表示及其提取

定义 2. 左值符号 *S* 的统一表示(*L*-value uniform representation)LVUR(*S*)=($Var,Of_1,Of_2,...,Of_n$) 表示 *S* 对应 $*(...*(*(\&Var+Of_1)+Of_2)+...)+Of_n$ 的存储空间,其中 Var 对应 C 程序中定义的变量,$Of_1,Of_2,...,Of_n$ 是内存地址偏移量序列.

比如对于 C 代码中的结构 *S* 和变量 *v*,*p*.

```
struct S {
    int i;
    struct S * s;
} v, * p;
```

左值符号 LVUR(*p*)=(*p*,0),对应*(&*p*+0)的存储空间;LVUR(*v*.*s*)=(*v*,sizeof(*int*)),对应*(&*v*+sizeof(*int*))的存储空间;*p*→*s* 和 *(*p*+sizeof(*int*)) 都对应 *(*(&*p*+0)+sizeof(*int*)) 的存储空间,因此具有相同的统一表示(*p*,0,sizeof(*int*)).由于在 C 程序中变量占用的存储空间在其整个生命周期中都是稳定不变的.因此,在此期间的所有执行文境中由该变量衍生出来的具有相同统一表示的左值符号都对应同一存储空间.之所以在此引入左值符号的统一表示,是为了区别处理 C 语言特性导致的静态别名现象和代码执行过程中传值操作引起的动态别名现象.继续参考前例,由于 C 语言中指针算术操作的存在,左值符号 *p*→*s* 和*(*p*+sizeof(*int*))在变量 *v*,*p* 的生命周期中都具有别名关系,如果能在此后的分析中统一地使用它们相同的统一表示,可以屏蔽这种静态别名现象,简化

后继对于动态别名关系的考察.由于别名现象会影响值等价类空间的改变,因此在维护值等价类空间时,这种对于动态别名关系的跟踪是必不可少的.

在没有传值操作的显式作用下,我们默认具有不同统一表示的左值符号对应不同的内存空间,也不存在等值关系.之所以一直强调变量的生命周期,是由于并非在所有具有相同统一表示的左值符号都对应相同的存储空间.以如下涉及 FreeBSD 锁机制的 C 代码为例:

```
void foo() {
    struct mtx mutex;
    …
}
```

当在不同执行文境中调用函数 foo 时,尽管这几次调用过程中左值符号 *mutex* 都具有相同的统一表示(*mutex*,0),但它们由于变量 *mutex* 处于不同的生命周期可能会对应不同的存储空间,事实上,开发人员也并不期待这些临界区之间存在任何联系.因此,在左值符号的统一表示中,变量的不同生命周期应该能够区别,为此,需要在 Var 条目中附加其作用域的信息.但为了讨论方便,下文涉及的统一表示都省略了这些附加信息,我们假设可以透明地依据 Var 条目区分出它所处的生命周期,而事实上,我们的实现通过维护作用域栈达到了这一效果.

由于左值符号是我们关注的指针算术操作的一部分,因此其统一表示可以和指针算术操作表达式的统一表示一起定义.之所以单独介绍,主要是为了说明如何基于 GCC4 的中间代码判定传值操作涉及的 C 语言构件是否为左值符号以及对于左值符号如何提取其统一表示.由于 GCC4 的中间代码——语法树中包含内存地址偏移量的信息且提供了可以用于识别指针算术操作的表达式类型信息,因此我们可以通过遍历 C 语言构件对应的语法树完成上述任务,递归地实现该功能的伪码如下所示:

```
LVUR(tree t,String & variable,vector⟨int⟩ & offsets,int & depth=0){
  switch (TREE_CODE(t)){
    case VAR_DECL:
      variable=string(IDENTIFIER_POINTER(DECL_NAME(t)));depth++;break;
    case COMPONENT_REF:
    case BIT_FIELD_REF:
      LVUR(TREE_OPERAND(t,0),variable,offsets,depth);
      offsets[depth]+=TREE_INT_CST_LOW(TREE_OPERAND(t,2));break;
    case INDIRECT_REF:
    case ALIGN_INDIRECT_REF:
      LVUR(TREE_OPERAND(t,0),variable,offsets,depth);depth++;break;
    case MISALIGNED_INDIRECT_REF:
      LVUR(TREE_OPERAND(t,0),variable,offsets,depth);depth++;
      offsets[depth]+=TREE_INT_CST_LOW(TREE_OPERAND(t,1));break;
    case ARRAY_REF:
    case ARRAY_RANGE_REF:
      LVUR(TREE_OPERAND(t,0),variable,offsets,depth);
      offsets[depth]+=TREE_INT_CST_LOW(TREE_OPERAND(t,1))*\
        TREE_INT_CST_LOW(TREE_OPERAND(t,3));break;
    case MODIFY_EXPR:
      LVUR(TREE_OPERAND(t,0),variable,offsets,depth);break;
    case PLUS_EXPR:
    case MINUS_EXPR:
```

```
        if(TREE_TYPE(t)=POINTER_TYPE){
          LVUR(TREE_OPERAND(t,0),variable,offsets,depth);
          offsets[depth]+/–=TREE_INT_CST_LOW(TREE_OPERAND(t,1));
        }
        break;
      case ADDR_EXPR:
        LVUR(TREE_OPERAND(t,0),variable,offsets,depth);depth––;break;
      default:
        return;
    }
  }
```

如果语法树 t 对应左值符号,其统一表示中的 Var 条目通过 *variable* 返回,偏移量序列通过 *offsets* 返回;*variable* 的返回值为空,表示 t 对应的并非左值符号.

3.3 指针算术操作表达式的统一表示及其相关操作、性质

定义 3. 指针算术操作表达式 Exp 的统一表示(pointer numeric operation uniform representation) PNOUR(Exp)= $((Var,Of_1,Of_2,...,Of_n),Int)$,对应形如 S+Int 的表达式,其中 S 为左值符号,LVUR(S)= $(Var,Of_1,Of_2,...,Of_n)$.

仍然以前文中定义的结构 S 和变量 v,p 为例,PNOUR(&v)=((v),0),PNOUR(p+sizeof(*int*))=((p,0),sizeof(*int*)). 由于指针算术操作表达式 Exp 对应形如"左值符号+整值"的表达式,因此可以通过简单扩展第 3.2 节中的递归函数 LVUR 来实现从语法树中提取指针算术操作表达式统一表示的功能.

由于在还原值等价类空间时我们必须考虑传值操作引入的动态别名现象的影响,因此,在具体介绍其实现之前,首先需要讨论指针算术表达式的统一表示相关的操作和性质.

定义 4. 如果 PNOUR(Exp)= $((Var,Of_1,Of_2,...,Of_n),Int)$,

PNOUR(Exp)=PNOUR(Exp)= $((Var,Of_1,Of_2,...,Of_n,Int),0)$;

此外,如果 n>0 且 Int=0,&PNOUR(Exp)=PNOUR(&Exp)= $((Var,Of_1,Of_2,...,Of_{n-1}),Of_n)$.

比如 PNOUR(p+sizeof(*int*))=PNOUR((p,0),sizeof(*int*)), 则 PNOUR($p\rightarrow$s)=PNOUR(*(p+sizeof(*int*))=((p,0,sizeof(*int*),0).由于 C 代码中&操作符只能作用于左值符号,因此只有当 PNOUR(Exp)中的 n>0 且 int=0 时,PNOUR(&Exp)才有意义.比如 PNOUR(v.s)=((v,sizeof(*int*),0),则 PNOUR(&v+sizeof(*int*))=PNOUR(&v.s)=((v),sizeof(*int*));而 PNOUR(&(&v+sizeof(*int*)))是不存在的.

定义 5. 如果指针算术操作表达式 A 和 B 满足 PNOUR(A)= $((AVar,AOf_1,AOf_2,...,AOf_{An}),IntA)$,PNOUR($B$)= $((BVar,BOf_1,BOf_2,...,BOf_{Bn}),IntB)$ 且 $AVar=BVar, An\le Bn, AOf_i=BOf_i(i=1...An)$,我们称指针算术操作表达式 B 在等值分析中受指针算术操作表达式 A 的影响,记作 A INF B.

比如,由于 PNOUR(p+sizeof(*int*))=((p,0),sizeof(*int*)),PNOUR($p\rightarrow i$)=((p,0,0),0),因此,p+sizeof(*int*) INF $p\rightarrow i$,事实上,p+sizeof(*int*)的取值发生改变必然源于 p 取值的改变,而 $p\rightarrow i$ 对应的存储空间也会随之改变,它涉及的等值关系必然会受到影响.

定义 6. 如果指针算术操作表达式 A 和 B 满足 PNOUR(A)= $((AVar,AOf_1,AOf_2,...,AOf_{An}),IntA)$,PNOUR($B$)= $((BVar,BOf_1,BOf_2,...,BOf_{Bn}),IntB)$ 且 A INF B,

当 An=Bn 时,PNOUR(B)–PNOUR(A)= $(IntB-IntA)$;

当 An<Bn 时,PNOUR(B)–PNOUR(A)= $(BOf_{An+1}-IntA, BOf_{An+2},...,BOf_{Bn}, IntB)$;

我们称 PNOUR(B)–PNOUR(A)的结果为差值序列(difference value sequence),记作 DVS.

我们仍然以指针算术操作表达式 p+sizeof(*int*)和 $p\rightarrow i$ 为例,PNOUR($p\rightarrow i$)–PNOUR(p+sizeof(*int*))= (–sizeof(int),0).

定义 7. 如果指针算术操作表达式 A 满足 PNOUR(A)= $((AVar, AOf_1, AOf_2,..., AOf_{An}), IntA)$,

那么,PNOUR(A)+(Int)= $((AVar, AOf_1, AOf_2,..., AOf_{An}), IntA + Int)$;

而 PNOUR(A)+ $(Int_1, Int_2,..., Int_n) = ((AVar, AOf_1, AOf_2,..., AOf_{An}, IntA + Int_1, Int_2,..., Int_{n-1}), Int_n)(n > 1)$.

比如 PNOUR(p+sizeof(int))=(p,0),sizeof(int)),与前例中的 DVS(−sizeof(int),0)的求和运算结果为((p,0,0),0)对应指针算术操作表达式 $p \rightarrow i$.

定理 1. 如果算术操作表达式 A 和 B 满足 A INF B,当 A 与算术操作表达式 C 建立等值关系时,B 与 PNOUR(C) +(PNOUR(B)−PNOUR(A))对应的指针算术操作表达式也会建立等值关系.

由于篇幅所限,在此省略该定理的证明过程.仍然以指针算术操作表达式 p+sizeof(int)和 $p \rightarrow i$ 为例,当 p+sizeof(int)与 struct S** 类型的 spp 建立等值关系时,$p \rightarrow i$ 应与 PNOUR(spp)+(PNOUR($p \rightarrow i$)−PNOUR(p+sizeof(int)))=((spp,0,−sizeof(int)),0)对应的*(spp−sizeof(int))建立等值关系.

在处理传值操作的过程中,我们可以依据上述定义和定理找到值等价类空间中所有等值关系受到传值操作影响的指针算术操作表达式并更新它们所对应的值等价类.

3.4 还原值等价类空间

在值等价类空间中,我们会维护一组值等价类实例,每个实例都包含若干具有等值关系的指针算术操作表达式的统一表示.我们会依据从执行路径中提取出的传值操作二元组⟨$Exp1$,$Exp2$⟩在值等价类实例间移动受到影响的统一表示,达到记录变量符号间等值关系的目的,实现该功能的伪码如下所示:

```
Update(EXP E1,EXP E2){
  //计算值等价类空间中和 E1 对应同一存储空间的统一表示
  set⟨PNOUR⟩ E1Aliases;
  foreach(PNOUR Pnour in VEC Space)
    if(Pnour INF PNOUR(&E1))
      foreach(PNOUR Equality in VEC containing Pnour)
          insert*(Equality+(PNOUR(&E1)−Pnour)) into E1Aliases;
  //计算值等价类空间中受影响的统一表示
  map⟨DVS,set⟨PNOUR⟩⟩ AffectedPNOURs;
  foreach(PNOUR E1Alias in E1Aliases)
    foreach(PNOUR Pnour in VEC Space)
      if(E1Alias INF Pnour)
        insert Pnour into AffectedPNOURs[Pnour−E1Alias];
    //更新值等价类空间中受影响的统一表示所对应的值等价类
    foreach(DVS Dvs in the keys of AffectedPNOURs){
      VEC Vec=new VEC;
      if(E2 is Pointer Numeric Operation)
        if((PNOUR (E2)+Dvs) in VEC Space)
          Vec=VEC containing (PNOUR (E2)+Dvs);
      foreach(PNOUR Pnour in AffectedPNOURs [Dvs])
        remove Pnour from VEC containing Pnour; insert Pnour into Vec;
    }
}
```

由于 C 语言支持结构指针的递归定义,所以我们对于可能涉及的指针算术操作表达式采取一种延迟加入的策略,只有在必要时才将其对应的统一表示加入到值等价类空间中.因此,在查询时有可能变量符号对应的值等价类已经存在,但其统一表示却还尚未加入值等价类空间.我们应当充分挖掘已有的信息,尝试为变量符号统一表示寻找其所在的值等价类,当该值等价类确实不存在时再为其分配新的实例,实现该功能的伪码如下所示:

```
VEC Inquire(EXP E1){
```

```
    if(E1 is not Pointer Numeric Operation)
      return NULL;
    if(PNOUR(E1) in VEC Space)
      return VEC containing PNOUR(E1);
    //寻找值等价类空间中和 E1 具有等值关系的统一表示所在的值等价类
    foreach(PNOUR Pnour in VEC Space)
      if(Pnour INF PNOUR(E1))
        foreach(PNOUR Equality in VEC containing Pnour)
          if((Equality+(PNOUR(E1)–Pnour)) in VEC Space){
            insert PNOUR(E1) into VEC containing (Equality+(PNOUR(E1)–Pnour));
            return VEC containing (Equality+(PNOUR(E1)–Pnour));
          }
    //分配新的值等价类实例
    VEC Vec=new VEC;
    insert PNOUR(E1) into Vec;
    return Vec;
}
```

4 实 验

我们基于 GCC4 设计实现了如图 4 所示的 C 代码静态分析工具.其中等值分析模块依据 GCC4 前端提取的传值操作还原值等价类空间.API 一致性检验模块依据等值分析模块提供的等值关系,将前端分发过来的以变量符号为占位符的 API 调用转化为以值等价类为占位符的语义,以此为基础分析代码中 API 的使用情况是否符合有限自动机模型描述的规范.

为了对基于值等价类空间的等值分析方法进行检验,我们尝试考察了 FreeBSD 6.0 内核中锁机制的使用情况.我们通过阅读 FreeBSD 的相关文档(包括 FreeBSD Architecture Handbook Chapter 2 Locking Notes 以及 FreeBSD Kernel Developer's Manual MUTEX(9),LOCK(9))了解了 *mutex* 相关 API 的使用规范.由于 *mutex* 允许递归加锁操作,所以我们主要关注"对未加锁的临界区进行解锁"和"临界区加锁后未解锁"这两类问题.为此,我们建立了如图 5 所示的有限自动机模型.如果有值等价类实例最终停留在状态 Unlock Before Lock,表明存在"对未加锁的临界区进行解锁"的情形,而停留在 Locked 或 Spin Locked,则表明存在"临界区加锁后未解锁"的情形.静态分析工具会记录值等价类实例涉及的所有独立语义的信息,通过回溯这些信息我们可以很快地定位引起问题的原因.

尽管分析结果中存在为数不少的误报,我们也在 FreeBSD 6.0 内核的驱动部分发现了几个真实存在的问题.这些问题已经提交给 FreeBSD 社区并得到了确认,详细信息可见 http://www.freebsd.org/cgi/query-pr.cgi?pr=100046 和 http://www.freebsd.org/cgi/query-pr.cgi?pr=107944.对于出现的误报,我们进行了仔细的分析,发现主要是由于 API 一致性检验模块没有对分支条件进行考察造成的,许多被判定存在问题的执行路径实际都是不可达路径.

5 结论和工作展望

我们针对代码静态分析中常见的问题设计了基于值等价类空间的等值分析方法.该方法通过考察实际软件系统中的 API 一致性得到了验证.引入基于该方法实现的等值分析模块,可以使后继的 API 一致性分析不必处理代码中的传值操作,从而将注意力更多地集中到其关注的以数值为论域的时序性质上.在我们分析 FreeBSD 6.0 内核代码的过程中出现了许多由分支条件引起的误报,考虑到使用标志位来指导 API 的调用是比较常见的情形,我们会在接下来的研究中借鉴 SLAM[6]和 BLAST[7]探讨针对分支条件的分析方法.

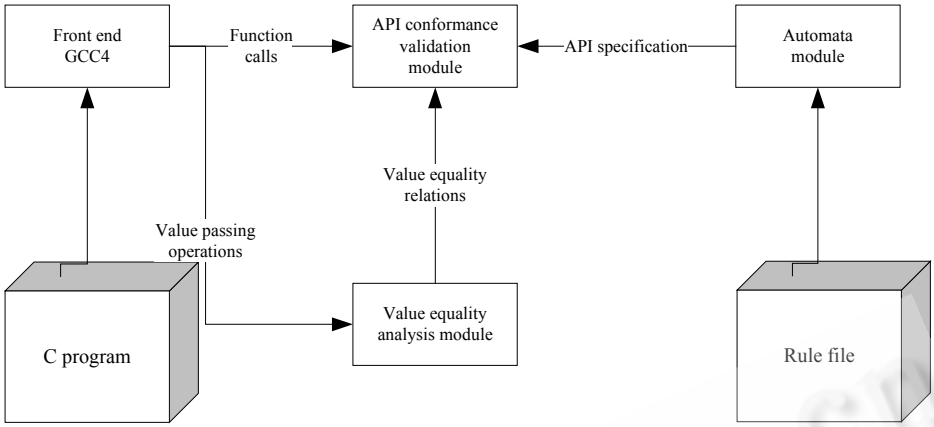

Fig.4 Architecture of the C program static analyzer

图 4 C 代码静态分析工具的架构

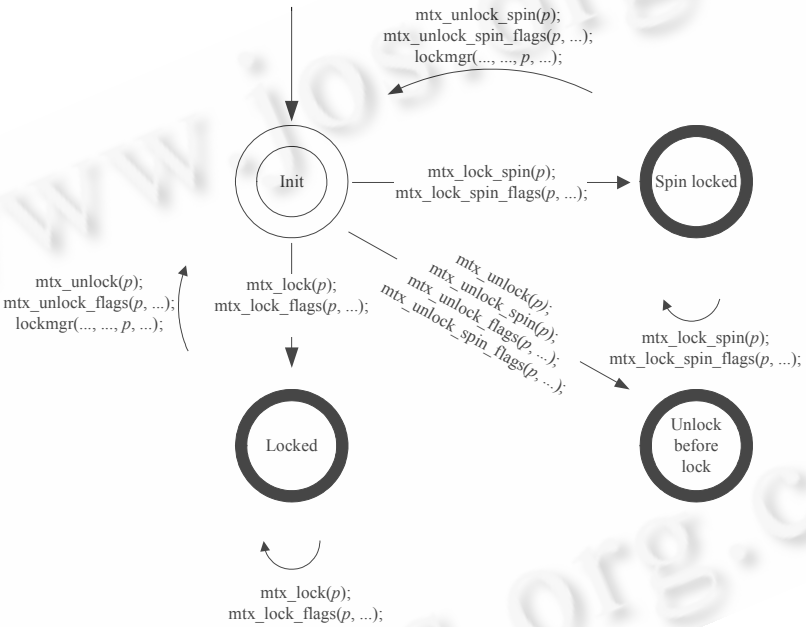

Fig.5 Finite automaton describing the usage rules of *mutex* in FreeBSD

图 5 描述 FreeBSD 内核中锁机制使用规范的有限自动机模型

致谢 在此我们向对本文的工作给予支持和建议的同行,尤其是中国科学院软件研究所基础软件中心参与文中 C 代码静态分析工具设计开发的曲富平和尹华祥表示感谢.

References:

[1] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In: Proc. of the 9th ACM Conf. on Computer and Communications Security. New York: ACM, 2002. 235−244. http://portal.acm.org/citation.cfm?id=894167&dl=ACM&coll=GUIDE&CFID=78554003&CFTOKEN=12830075

[2] Chen H, Shapiro J. Using build-integrated static checking to preserve correctness invariants. In: Proc. of the 11th ACM Conf. on Computer and Communications Security. New York: ACM, 2004. 288−297. http://portal.acm.org/citation.cfm?id=1030083.1030122&coll=GUIDE&dl=ACM&CFID=78554003&CFTOKEN=12830075

[3] Zhang XL, Edwards A, Jaeger T. Using CQUAL for static analysis of authorization hook placement. In: Proc. of the 11th USENIX Security Symp. Berkeley: USENIX Association, 2002. 33−48 http://portal.acm.org/citation.cfm?id=647253.720279&coll=GUIDE&dl=GUIDE&CFID=2159995&CFTOKEN=54693925

[4] Edwards A, Jaeger T, Zhang XL. Runtime verification of authorization hook placement for the Linux security modules framework. In: Proc. of the 9th ACM Conf. on Computer and Communications Security. New York: ACM, 2002. 225−234. http://portal.acm.org/citation.cfm?id=586110.586141&coll=GUIDE&dl=GUIDE&CFID=2159995&CFTOKEN=54693925

[5] Engler D, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. of the 4th Conf. on Symp. on Operating System Design and Implementation. Berkeley: USENIX Association, 2000. 1−16. http://portal.acm.org/citation.cfm?id=1251229.1251230&coll=GUIDE&dl=GUIDE&CFID=2159995&CFTOKEN=54693925

[6] Ball T, Rajamani S. The SLAM project: Debugging system software via static analysis. In: Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM, 2002. 1−3.

[7] Henzinger T, Jhala R, Majumdar R, Sutre G. Software verification with BLAST. In: Proc. of the 10th Int'l Workshop on Model Checking of Software. London: Springer-Verlag, 2003. 235−239. http://www.springerlink.com/content/r5xy1apqk8fpxf6a/

[8] Henzinger T, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In: Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM, 2002. 58−70.

[9] Henzinger T, Jhala R, Majumdar R, Necula GC, Sutre G. Temporal-Safety proofs for systems code. In: Proc. of the 14th Int'l Conf. on Computer Aided Verification. London: Springer-Verlag, 2002. 526−538.

[10] Evans D, Larochelle D. Improving security using extensible lightweight static analysis. IEEE Software, 2001,19(1):42−51.

[11] Foster JS, Fahndrich M, Aiken A. A theory of type qualifiers. In: Proc. of the ACM SIGPLAN 1999 Conf. on Programming Language Design and Implementation. New York: ACM, 1999. 192−203.

[12] Foster JS. Type qualifiers: Lightweight specifications to improve software quality [Ph.D. Thesis]. Berkeley: University of California, 2002.

[13] Alur R, Cerny P, Madhusudan P, Nam W. Synthesis of interface specifications for Java classes. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM, 2005. 98−109.

[14] Das M, Lerner S, Seigle M. ESP: Path-Sensitive program verification in polynomial time. In: Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation. New York: ACM, 2002. 57−68.

[15] Ball T, Majumdar R, Millstein T, Rajamani S. Automatic predicate abstraction of C programs. In: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation. New York: ACM, 2001. 203−213.

[16] Xia YM, Luo J, Zhang MX. Security vulnerability detection study based on static analysis. Computer Science, 2006,33(10):279−282 (in Chinese with English abstract).

[17] Jiang JQ, Luo H, Zeng QK. Research on program vulnerabilities analysis and program security protection technologies. Computer Applications and Software, 2007,24(1):19−23 (in Chinese with English abstract).

[18] Zhang MJ, Luo J. Pointer analysis algorithm in static buffer overflow analysis. Computer Engineering, 2005,31(18):41−43 (in Chinese with English abstract).

附中文参考文献:

[16] 夏一民,罗军,张民选.基于静态分析的安全漏洞检测技术研究.计算机科学,2006,33(10):279−282.

[17] 蒋剑琴,罗宏,曾庆凯.程序缺陷分析与安全保护技术研究.计算机应用与软件,2007,24(1):19−23.

[18] 张明军,罗军.缓冲区溢出静态分析中的指针分析算法.计算机工程,2005,31(18):41−43.

孟策(1981－),男,天津人,博士生,主要研究领域为系统软件安全技术,代码静态分析技术.

罗宇翔(1983－),男,硕士,主要研究领域为系统软件安全技术,代码静态分析技术.

贺也平(1962－),男,博士,研究员,博士生导师,主要研究领域为可信计算技术.