软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(3):472−488 [doi: 10.13328/j.cnki.jos.004419]

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

一种场景敏感的高效错误检测方法*

衷璐洁[1,2,3], 霍 玮[4], 李 龙[1], 李 丰[1], 冯晓兵[1], 张兆庆[1]

[1](中国科学院 计算技术研究所 计算机体系结构国家重点实验室,北京 100190)

[2](首都师范大学 信息工程学院,北京 100089)

[3](中国科学院大学,北京 100049)

[4](中国科学院 信息工程研究所 第五研究室,北京 100093)

通讯作者: 衷璐洁, E-mail: zhonglujie@ict.ac.cn

摘 要: 定值-引用类错误是一类非常重要且常见的错误.当前,对这类错误的检测很难同时达到高精度和高可扩展性.通过合理组合敏感和不敏感的检测方法并控制两类方法的实施范围,可以同时达到高检测精度和高可扩展性.提出一种新颖的场景敏感的检测方法,该方法根据触发状态对潜在错误语句分类,识别不同类别语句的触发场景并实施不同开销的检测,在不降低精度的同时最小化检测开销.设计了一个多项式时间复杂度的流敏感、域敏感和上下文敏感的场景分析以进行分类,并基于程序依赖信息识别触发场景,仅对必要的触发场景实施路径敏感的检测.为上述方法实现了一种原型系统——Minerva.通过使用空指针引用错误检测为实例研究以及总代码规模超过 290 万行,最大单个应用超过 200 万行的应用验证,用例实验结果表明,Minerva 的平均检测时间比当前先进水平的路径敏感检测工具 Clang-sa 和 Saturn 分别快 3 倍和 46 倍.而 Minerva 的误报率仅为 24%,是 Clang-sa 和 Saturn 误报率的 1/3 左右,并且 Minerva 未发现漏报已知错误.上述数据表明,所提出的场景敏感的错误检测方法可同时获得高可扩展性和高检测精度.

关键词: 定值-引用错误;路径敏感错误检测;错误目标触发场景;场景敏感;程序分析

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 衷璐洁,霍玮,李龙,李丰,冯晓兵,张兆庆.一种场景敏感的高效错误检测方法.软件学报,2014,25(3):472−488. http://www.jos.org.cn/1000-9825/4419.htm

英文引用格式: Zhong LJ, Huo W, Li L, Li F, Feng XB, Zhang ZQ. Efficient scene-sensitive fault detection approach. Ruan Jian Xue Bao/Journal of Software, 2014,25(3):472−488 (in Chinese). http://www.jos.org.cn/1000-9825/4419.htm

Efficient Scene-Sensitive Fault Detection Approach

ZHONG Lu-Jie[1,2,3], HUO Wei[4], LI Long[1], LI Feng[1], FENG Xiao-Bing[1], ZHANG Zhao-Qing[1]

[1](State Key Laboratory of Computer Architecture, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100190, China)

[2](Information Engineering College, Capital Normal University, Beijing 100089, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

[4](No.5 Research Laboratory, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100049, China)

Corresponding author: ZHONG Lu-Jie, E-mail: zhonglujie@ict.ac.cn

Abstract: Def-Use faults are a very important and common type of faults. The state-of-the-art detection schemes for such faults still hardly achieve both preciseness and scalability. This paper applies the idea of combining the sensitive and insensitive detection approaches and deploying the effective range of the two approaches to achieve both high detection scalability and high precision. The

* 基金项目: 国家自然科学基金(61100011, 61202055, 61303053); 国家高技术研究发展计划(863)(2012AA010901); 国家自然科学基金创新研究群体科学基金(60921002)

收稿时间: 2012-10-08; 修改时间: 2013-03-20; 定稿时间: 2013-04-07

study results in a new scene- sensitive detection strategy based on a classification scheme on statements that contain potential faults. The key idea is to classify these statements into different categories based on how a potential fault in these statements might be triggered. It uses polynomial flow-, field- and context-sensitive summary based scene analysis to do the classification and identifies triggering scenes based on program dependence information. Different detection schemes with different amount of overheads are then applied to different categories and thus reducing the overall overhead and achieving a higher scalability. The path-sensitive detection schemes are only performed on the necessary triggering scenes. The proposed approach is implemented in a prototype system, called Minerva. Using null pointer dereference fault detection as an example and verifying the approach through applications whose total code size exceed 2.9 million lines (one application exceeds 2 million lines), the experimental results show that the average detection time of Minerva is 3× and 46× faster than the two state-of-the-art path-sensitive detection tools, Clang-sa and Saturn, respectively. The false positive rate of Minerva is 24%, which is also a third of that of Clang-sa and Saturn's. There is no false negative on the known faults. The results show that the proposed scene-sensitive fault detection approach can achieve both high scalability and high accuracy.

Key words: def-use fault; fault detection; sink triggering scene; scene-sensitive; program analysis

定值-引用类错误通常很容易发生但较难精确地检测.代表性的定值-引用类错误包括空指针引用、未赋值引用、除零错、缓冲区溢出等.这类错误难以精确检测,是因为这些错误中由错误源到达错误目标的路径往往很长且涉及的路径数众多.不仅规模大的应用程序包含大量的执行路径,即便是小规模的程序,仍有可能具有大量的路径数.例如 SPEC CPU2000 中的 164.gzip,其程序代码规模仅为 8 000 行,但涉及的路径数却超过了 3.49E+11[1].

一直以来,高可扩展性和高检测精度都是静态错误检测所追求的目标.为了获得高检测精度,有很多方法采用了路径敏感的检测策略,但这些方法在可扩展性上的不足严重影响了它们的实用性.为了提高可扩展性,一些路径敏感的检测方法引入了需求驱动的策略,但仍然存在这样一些问题:1) 检测的可扩展性提高了,但检测精度却有所降低.例如在 Clang-sa[2]中,为了减少开销,它限制了所能处理的路径条件表达式的类型,因此导致了检测精度上的损失;2) 路径敏感的开销仍然很大.如文献[3−7]实施路径敏感检测的范围是整个程序中所有的潜在错误语句,但对于定值-引用类错误而言,潜在错误语句涉及的引用点数量众多,错误触发的场景也更为复杂,对于这样的情形,通常以切片为主要技术的需求驱动策略往往不够有效.

我们对 OpenSSH[8]和 Wireshark[9]等实用程序进行分析后发现:1) 有些错误在每条潜在执行路径上都会被触发; 2) 有些错误在任何潜在执行路径上都不会被触发;3) 有些错误可能在一些但不是全部的执行路径上才会被触发.基于上述发现,我们认为为了保证检测精度,有必要采用路径敏感的检测方法,但是没有必要对所有的潜在错误语句都采用路径敏感的检测方法.

如图 1 所示片段为例,该片段提取自 Wireshark[9],一个实际的网络协议分析应用,其代码超过了 200 万行.从错误源(可能引起错误的变量定值点)到错误目标(可能触发错误的变量引用点)的路径较长(第 4 行~第 36 行涉及大量代码,图中…代表省略的代码段).但图中第 34 行的宏 *DISSECTOR_ASSERT* 会阻止潜在的不安全值流向错误目标.这样,第 36 行在对 ie_item 变量进行解引用时不安全的值将不会到达.可是对于这样的情形,传统的路径敏感检测方法需要对图中标识为灰色的代码区域中各条潜在执行路径逐一进行路径敏感的分析,产生较长的路径条件约束并进行求解,但事实上,对于该例中的这种情形,如果能够识别宏 *DISSECTOR_ASSERT* 中的特殊控制流,即可判别第 36 行的危险值解引用不会到达,这样可以避免大量冗余的检测开销.同时值得一提的是,类似这样的在值引用之前进行检查的现象,在实际应用程序中并不少见.

基于以上分析,我们认为错误检测可以通过组合低开销的路径不敏感方法和高精度的路径敏感方法来同时获得高可扩展性和检测精度.在本文中,我们提出了一个可扩展的、高精度的错误检测框架,它基于潜在错误语句分类进行错误检测.考虑错误是否会被触发的场景,我们将程序中所有的潜在错误语句分为一定触发(must-trigger)、一定不触发(must-not-trigger)和可能触发(trigger-unknown)这 3 类,同时设计一个多项式时间的流敏感、域敏感和上下文敏感的传播引擎去实施分类.在分类后,对位于潜在错误语句中的不同类别的错误目标应用合适的、高效的检测算法进行处理.对于一定触发和一定不触发的潜在错误语句,通过低开销的路径不敏感方法完成检测;而对于可能触发的潜在错误语句,采用路径敏感的检测方法.通过这样的方式,我们可以实

现路径敏感与路径不敏感检测的合理组合.由于在一个程序中路径不敏感和路径敏感的检测方法的应用范围是互补的,上述的组合方式在拓展路径不敏感检测有效范围的同时,缩小了路径敏感检测的范围,可以实现总开销的合理调控.

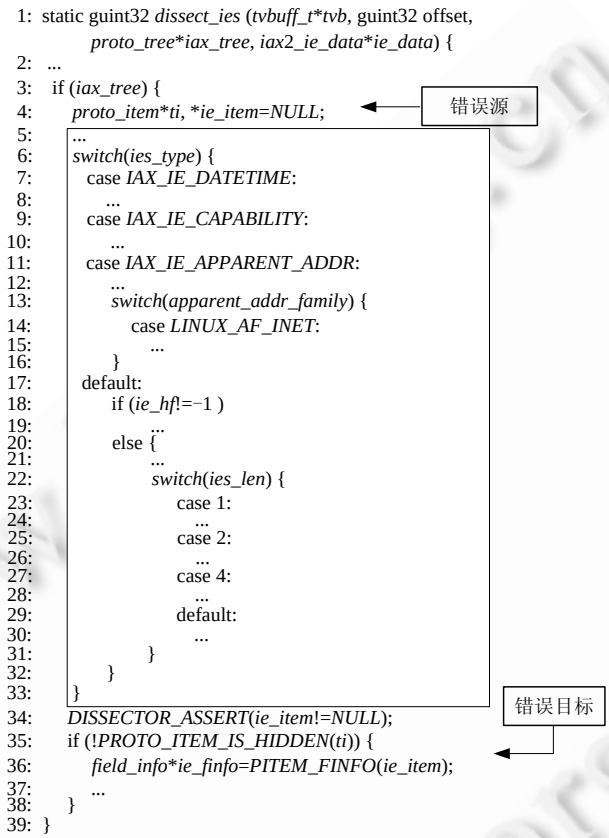


Fig.1 A motivating example (from Wireshark)

图 1 研究动机示例(选自 Wireshark)

本文贡献如下:

1) 提出利用组合路径敏感和不敏感检测的策略,以同时达到错误检测高可扩展性和高检测精度的目标.通过拓广一个程序中路径不敏感检测方法的有效范围,缩小需要进行路径敏感检测的范围.并基于上述想法提出一种场景敏感的检测方法,该方法基于按不同触发场景对程序中潜在错误语句进行分类的思想,将潜在错误语句分为:一定触发、一定不触发和可能触发这 3 类.通过路径不敏感的方法完成场景分类以及一定触发和一定不触发的潜在错误语句的处理,仅对可能触发的潜在错误语句利用路径敏感的方法进行检测;

2) 在开放源码编译器 Open64[10]中实现了上述方法的原型系统 Minerva,并在应用广泛、超过 290 万行规模的代码上对 Minerva 进行了验证.通过将空指针引用错误检测作为实例研究,验证了场景敏感检测方法的有效性.结果表明,对于本文的实验用例,Minerva 的平均检测时间分别比当前先进水平的路径敏感检测工具 Clang-sa 和 Saturn 快了约 3 倍和 46 倍.平均误报率仅为 24%,是 Clang-sa 和 Saturn 平均误报率的 1/3 左右,且没有漏报已知错误.

第 1 节介绍术语.第 2 节阐述场景敏感检测方法的全貌及 Minerva 框架,并讨论如何对潜在错误语句进行分类,以及如何基于这样的分类进行相应的检测算法设计.第 3 节给出场景敏感检测效益模型.第 4 节是原型系统的实现.第 5 节给出实验结果,包括实验环境和实验用例.第 6 节是相关工作.最后对全文进行总结.

1 术 语

有一类重要的错误,它们通常表现为在变量的定值点分配给变量一个不安全(unsafe)的值,并且这个不安全的值会沿着至少一条执行路径流向该变量的引用点,从而触发一个错误.对这样一类错误,本文称为定值-引用类错误.定值-引用类错误中的引用点被称为错误目标(sink),相应定值点被称为错误源(source).

定义 1(触发场景). 给定变量 v 的一个引用点(记作 $USE(v)$),所有沿着控制流路径可以到达该引用点的定值点集合(记作 $DEFs(v)$)与给定引用点以及它们之间的路径所构成的控制流子图是一个触发场景.我们用触发场景图(triggering-scene graph,简称 TSG)对触发场景进行描述.将变量 v 的一个引用点 $USE(v)$的 TSG 记为 $TSG_{USE(v)}=\langle V,E\rangle$,其中,$V\subseteq V_{CFG}$,$DEFs(v)\subseteq V$,$USE(v)\in V$,$E\subseteq E_{CFG}$,$V_{CFG}$ 和 E_{CFG} 分别表示控制流图 CFG 的顶点集和边集.

定义 2(触发状态). 划分 3 种触发状态:一定触发、一定不触发和可能触发,定义如下:

- 一定触发(must-trigger)状态:在 TSG_v 上,如果$\forall s\in DEFs(v)$,定值 s 都是不安全的,那么称 $USE(v)$处于一定触发状态;
- 一定不触发(must-not-trigger)状态:在 TSG_v 上,如果$\forall s\in DEFs(v)$:
  (1) 定值 s 是安全的,或者
  (2) 定值 s 是不安全的,但从 s 到 $USE(v)$的值流路径不可行,
  那么称 $USE(v)$处于一定不触发状态;
- 可能触发(trigger-unknown)状态:在 TSG_v 上,如果$\exists s\in DEFs(v)$,定值 s 是安全的并且$\exists s'\in DEFs(v)$,定值 s'是不安全的且 s 和 s'到 $USE(v)$的值流路径可能可行,称 $USE(v)$处于可能触发状态.

一般地,若 $USE(v)$有不安全值到达,称 $USE(v)$所在的语句为潜在错误语句.

文中场景敏感的错误检测方法是指在执行错误检测时会区分不同的错误目标触发状态,即对每一种错误目标触发状态,场景敏感的错误检测方法会选择最合适开销的方法进行检测;相反地,一个场景不敏感的错误检测方法并不会在检测时区分不同的错误目标触发状态.

定值-引用类错误的检测问题本质上是一个广义的定值-引用流分析问题.受数据流分析框架[11]的启发,我们用格值模型描述定值-引用类错误的属性.该模型将定值-引用类错误的检测问题域组织成一个格,称为错误属性格.基于错误属性格的定义,我们将定值-引用类错误的检测问题变换为错误属性格值的计算与传播问题,并将负责错误属性格值计算与传播的部分称为传播引擎(propagation engine,简称 PE).

定义 3(错误属性格). 错误属性格(fault attribute lattice,简称 FAL)是一个代数系统,定义为 $FAL=(V_{FAL},F_{intersection},F_{union})$,其中,

- 集合 V_{FAL} 中的元素是具体错误类型的错误属性格值;
- V_{FAL} 中的底元素$\bot$表示初始的错误属性值,顶元素$\top$表示检测到的将要报告的错误属性值;
- 函数 $F_{intersection}:V_{FAL}\times V_{FAL}\to V_{FAL}$ 和 $F_{union}:V_{FAL}\times V_{FAL}\to V_{FAL}$ 分别是 FAL 中的最大下界和最小上界函数,它们实现将错误属性值计算映射为 FAL 上的格值操作;
- FAL 上的偏序关系对应错误属性格值的计算规则;
- FAL 值具有 3 种属性:安全、一定不安全和可能不安全.其中,安全的 FAL 值对应一定不触发的状态;一定不安全的 FAL 值对应一定触发状态;可能不安全的 FAL 值则需要进一步判断:若值流路径无法到达引用点,则对应一定不触发状态;否则,对应可能触发状态.

2 场景敏感的错误检测策略

2.1 错误触发场景

一般地,错误触发场景通常表现为如图 2 所示的 3 种典型形式:

(1) 对于如图 2(a)所示的一定触发场景,无论程序执行哪条从定值点到引用点的路径,都存在源自错误源

的不安全值到达错误目标;

(2) 对于如图 2(b)所示的一定不触发场景,因为在错误目标与错误源之间存在对值进行引用前检查的检查语句,即图中是否不安全?判定框,所以对于错误目标而言值流将不可达.这是一种常见的程序员安全编程习惯:在使用一个值,特别是指针值之前,先去检查该值是否是不安全的,若是不安全的,就会引导程序执行绕过引用点以避免错误的发生.这样的处理方式通常会改变正常的程序控制流,称为特殊控制流(special control flow,简称 SCF);

(3) 对于如图 2(c)所示的可能触发场景,错误源到错误目标的值流是否可行取决于条件(1),而这需要利用路径敏感的分析来判定.

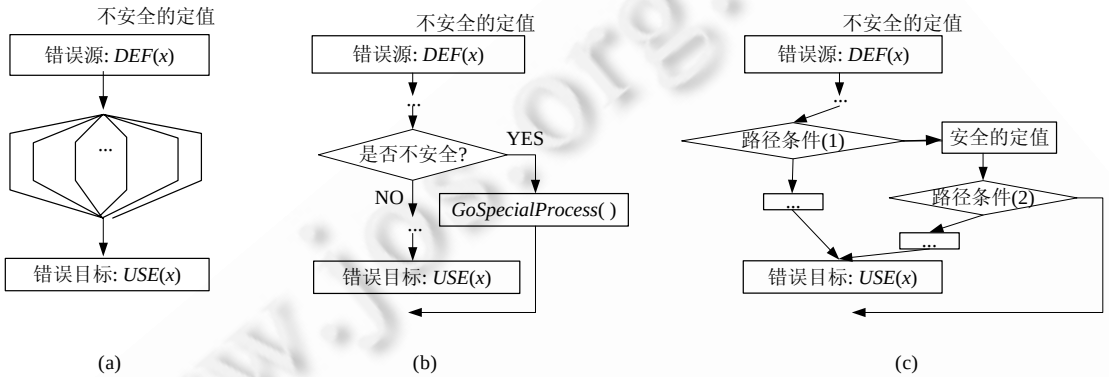


Fig.2 Typical sink-triggering scenes

图 2 典型的错误触发场景

我们统计了实验中所有程序内一定触发、一定不触发和可能触发的错误目标的数量,发现只有 6%的错误目标是可能触发的,其余均为一定触发或一定不触发的,并且一定不触发的错误目标比例高达 73%.这意味着高效的分类算法以及仅对可能触发的错误目标进行路径敏感的检测对达到在不降低精度的情况下提高检测效率的目标而言是有实际应用基础的.

2.2 Minerva检测框架

图3给出了Minerva检测框架的示意图.首先,针对具体的定值-引用类错误创建一种错误属性格(FAL)模型,将错误属性定义为格值,通过这种方式,传播引擎(PE)将错误检测问题变换为 FAL 值的计算与传播问题.随后,PE 以一种静态的、路径不敏感但上下文敏感的方式计算程序中的 FAL 值,对错误目标进行分类,并根据这些错误目标的 FAL 值属性进行如下处理:

1) 对于那些具有一定不安全属性的错误目标,即无论执行程序中的哪条路径它们都一定会被触发.对于这些错误目标,PE 会将它们归类为一定触发,并直接给出错误报告;

2) 对于那些具有可能不安全属性的错误目标,PE 会进一步判定这些错误目标是否是一定不触发的错误目标:如果是,意味着相应错误目标无论在哪条路径下都不会被触发,PE 将过滤它们;否则,相应的错误目标可能在某些路径上被触发,那么 PE 会将它们归类为可能触发的,然后把它们放到一个切片标准工作集中(该工作集在第 2.4 节进行介绍);

3) 最后,切片标准工作集中的每个元素都会被 Minerva 选择作为切片种子进行程序切片,然后在切片后的结果程序上实施路径敏感的检测.

通过潜在错误语句分类和仅在必要的程序切片结果上实施路径敏感的检测,我们可以避免因为采用统一的路径敏感检测方法处理所有错误目标而产生的不必要开销.

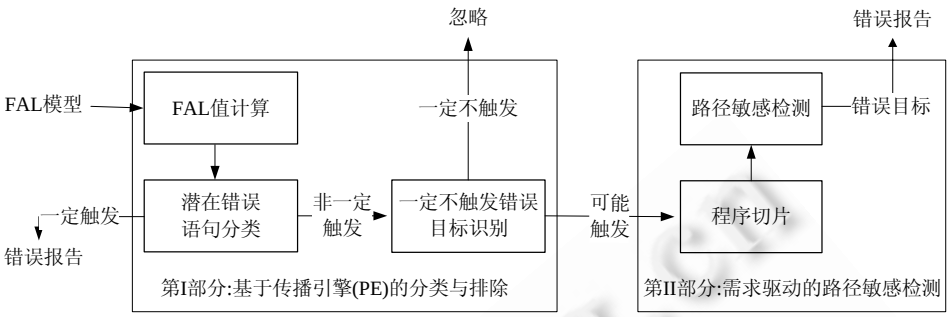


Fig.3 Minerva detection framework

图 3 Minerva 检测框架

2.3 FAL值计算及场景分类

2.3.1 FAL 值计算

Minerva 利用 PE 在错误属性格值计算与传播的过程中找出所有的错误目标.FAL 值计算由两部分组成:过程间基于摘要的上下文敏感的 FAL 值计算和过程内的流敏感和域敏感的 FAL 值计算.

过程间 FAL 值计算是一个两遍的、基于摘要的上下文敏感的计算过程.给定一个函数,为了使过程间 FAL 值的计算更加准确,我们将函数的输入和输出参数分别建模为引用输入(*Use_In*)和定值输出(*Def_Out*).其中,引用输入用于建模函数的输入参数,通常包括形式参数和全局量.定值输出用于建模函数的输出参数,通常包括返回值和函数的副作用.过程间上下文敏感的 FAL 值计算由 3 部分组成:将调用点处上下文错误属性格值传入给引用输入(*Map*(*ActualIn_VAL*(*cs*),*Use_In*(*cs*)))、函数体内的格值计算(*Intra_Compute*(f_{cs_callee}))以及将定值输出的错误属性格值传出给主调函数中调用点处(*Map*(*DefOut_*V_{AL}(*cs*),*Actual_Out*(*cs*))).其中,*ActualIn_* V_{AL}(*cs*)是在调用点 *cs* 处传入函数 f_{cs_callee} 的错误属性格值,*Use_In*(*cs*)是 f_{cs_callee} 的引用输入,*DefOut_*V_{AL}(*cs*)是在调用点 *cs* 由函数 f_{cs_callee} 传出的错误属性格值,*Actual_Out*(*cs*)是在调用点 *cs* 处被函数 f_{cs_callee} 返回值修改的变量或在函数 f_{cs_callee} 中修改的全局量.*Map*(V_{AL},*V*)表示将错误属性格值集合 V_{AL} 中的元素映射到变量集合 *V* 中对应变量.过程内 FAL 值计算 *Intra_Compute*(*f*)是流敏感和域敏感的,计算过程基于 SSA(静态单赋值)形式对函数中的语句逐条进行遍历.

2.3.2 错误目标触发场景分类算法

错误目标的场景分类在 FAL 值的计算与传播过程中完成.如果一个引用点被发现具有不安全的 FAL 值,PE 将会进一步判定其是否为一定不安全或可能不安全的 FAL 值.如果它是一定不安全的 FAL 值,Minerva 将会标识其为一定触发的错误目标,然后报告错误.否则,PE 会继续判断它是否是一定不触发的,这种判定在 USTSG 上完成.PE 会进行一定不触发的模式识别.图 2(b)描述了常见的一定不触发模式,它们通常由一条检查语句和一个 SCF 构成.

图 4 相应给出了图 2(b)中 *GoSpecialProcess*()可能出现的一些情形:根据改变控制流的方式,分为直接改变控制流和间接改变控制流两类,前者常常直接执行 return 或 exit,而后者通常将控制流改变语句包含在一个包装(wrapper)函数或宏中.图 5 给出了实际程序中为避免空指针引用或缓冲区溢出错误程序员编写的一些检查语句和 SCF 代码片段,其中,图 5(a)、图 5(b)是针对空指针引用而写的代码,图 5(a)在指针变量 *b* 为空指针时,会启用宏 *YY_FATAL_ERROR* 进行正常控制流改变;而图 5(b)在 *ap_server_root_relative* 函数的返回值为空时,直接 return 以改变正常的程序控制流.图 5(c)则是为了防止缓冲区溢出所编写的一些处理代码,其中,程序员通过条件 *t==tag+tagbuf_len* 来判断是否到达了缓冲区边界,如果到达就提前终止控制流以避免其后可能发生的缓冲区溢出错误.

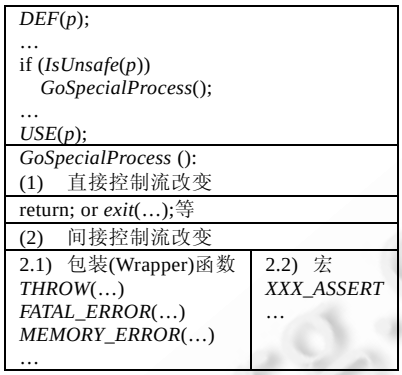


Fig.4 Typical form of GoSpecialProcess()

图 4 *GoSpecialProcess*()的常见形式

```
b=(YY_BUFFER_STATE)Matealloc(…);

if (!b)
    YY_FATAL_ERROR(…);
```

(a) 空指针引用 SCF(选自 Wireshark)

```
fspec=ap_server_root_relative(…);

if (!fspec) {
    ap_log_error(…);
    return NULL;
}
```

(b) 空指针引用 SCF(选自 Apache)

```
if (t==tag+tagbuf_len)
{
    *t=EOS;
    return NULL;
}
```

(c) 缓冲区溢出 SCF(选自 Apache)

Fig.5 SCF code segments example

图 5 实际 SCF 代码片段示例

为了更进一步验证危险值判断现象存在的普遍性,我们调研和分析了大量的实际应用程序源码(超过18000KLOC),包括 OpenSSH,Wireshark,Linux-kernel 等.结果发现,在我们的统计用例中,约每 1 000 行就有 8 行的空指针引用检查语句和 6 行的缓冲区溢出下标范围检查语句.具体的统计数据参见表 1.

Table 1 The number of unsafe value checking statements

表 1 危险值检查语句数统计

程序	规模(KLOC)	危险值检查语句数	
		空指针	缓冲区溢出
Sendmail	115	≥1082	≥233
Openssh	155	≥631	≥88
Apache	268.9	≥964	≥208
Wine	1 905	≥2600	≥122
Wireshark	2 383	≥1642	≥311
Linux-Kernal	13 500	≥133993	≥126867

在此值得一提的是,许多研究者在他们的研究中加入了对程序特征的考虑,例如在 Ngo 等人[12]的研究中,他们通过识别具有不同特征的 4 种模式来识别不可行的路径.在本文中,我们将对特殊的程序特征的考虑加入到了错误检测的工作中.

算法 1 给出了进行场景分类的算法,该算法在分类的同时检测一定触发的错误目标并识别一定不触发的错误目标.该算法通过区分不安全的 FAL 值的属性来完成场景分类.一般地,一定不触发的错误目标的识别方法是在 USTSG 中检查一个给定错误目标相对于错误源的所有控制节点(第 7 行~第 11 行),且只关注与错误相关的条件,例如寻找是否存在如图 2(b)所示的检查语句.如果存在,则继续查找是否存在 SCF.如果 SCF 也被找到了,那么当前的错误目标将会排除,因为它是一个一定不触发的错误目标(第 8 行~第 10 行).否则,检查下一个控制节点以继续寻找检查语句和 SCF.如果在所有的控制节点中都没有发现检查语句和 SCF,那么就将当前错误目标标识为可能触发的.在算法 1 中,我们将可能触发的错误目标及其错误源一并放入切片标准工作集中(第 12 行~

第 14 行).

算法 1. 错误目标的检测与排除.

```
1.  Procedure Detection_and_Exclusion(f)
2.    For each s in f
3.      If (s 中的一个引用点具有不安全的 FAL 值) Then
4.        If (FAL 值具有一定不安全属性) Then
5.          报告该错误;
6.        Else
7.          For each s 的控制节点
8.            If (检查语句和 SCF 成功匹配) Then
9.              过滤当前引用点;
10.           End If
11.         End For
12.         If (在所有控制节点中均没有成功匹配) Then
13.           将当前引用点及其定值点加入到一个切片标准工作集中;
14.         End If
15.       End If
16.     End If
17.   End For
18. End
```

2.4 可能触发错误的检测

2.4.1 方法描述

对于算法 1 所生成的包含所有可能触发的错误目标的工作集,需要使用路径敏感的信息进行判定.为了有效降低路径敏感的检测开销,在实施路径敏感检测前,先进行以可能触发的错误目标为切片种子的程序切片以获得最小的路径敏感检测输入集.可能触发的错误目标程序切片由两个阶段组成:一个是准备阶段,在该阶段,PE 收集所有可能触发的错误目标,为路径敏感检测阶段做准备;另一个是路径敏感检测阶段,为了缩小检测范围,我们仅对错误目标工作集中的可能触发的错误目标而不是程序中所有的潜在错误语句进行路径敏感的检测.并且对于切片标准工作集中的每一个错误目标,我们也不是在全程序范围内进行检查,而是仅检查那些与该错误目标及相应错误源相关的语句.上述两种做法可以极大地减小路径敏感检测的输入集.

需要指出的是,对于可能触发的错误检测,我们依据控制流图(CFG)和 SSA 上的值流信息就可以进行切片,没有必要再去构造程序依赖图(PDG),这样可以进一步减少开销.

2.4.2 可能触发错误目标切片(trigger-unknown sink slicing,简称 TUSS)

为了提取那些可能影响错误源、被错误源影响以及影响错误目标的语句,在算法 2 中,我们首先对错误目标实施后向切片获得语句集合 S_{BT},随后对错误源实施前向切片得到语句集合 S_{FS},之后求这两个集合的交集 S_{TS}.然后对错误源实施后向切片,最后将其结果与 S_{TS} 求并集,所得结果即为所求.该算法中,*DoBackSlicing*(*x*,*y*)表示以 *y* 为切片标准对 *x* 实施后向切片,*DoForwardSlicing*(*x*,*y*)表示以 *y* 为标准对 *x* 实施前向切片.

集合 $S_{path\text{-}sensitive\text{-}detection}$ 是 TUSS 算法的输出,该集合是针对错误检测而言处于可能触发状态的潜在错误语句最相关的语句集合,在后面的工作中我们将对它们实施路径敏感的检测,换言之,该集合是 Minerva 中路径敏感检测部分的输入.

算法 2. 可能触发错误目标制导切片.

```
1.  Procedure TUSS(P,Workset_SlicingCriteria)
2.    For each (S_source,S_sink) in Workset_SlicingCriteria
```

3. S_{BT}←*DoBackSlicing*(P,S_{sink})
4. S_{FS}←*DoForwardSlicing*(P,S_{source})
5. S_{TS}←S_{BT}∩S_{FS}
6. S_{BS}←*DoBackSlicing*(P,S_{source})
7. S_{TB}←S_{TS}∪S_{BS}
8. $S_{path\text{-}sensitive\text{-}detection}$←$S_{TB}$
9. End for
10. End

2.5 达到高可扩展性的原因

1) 区分不同的错误目标触发场景,并基于潜在错误语句分类.

我们的检测策略采用不同开销的路径不敏感和路径敏感的检测方法组合,对于一定触发或一定不触发的错误目标,我们使用低开销的路径不敏感的方法进行检测和识别.仅对可能触发的错误目标,才采用路径敏感的检测方法,并通过 TUSS 有效控制开销.

2) 使用多项式时间的 FAL 值计算和传播算法实现对一定触发的错误目标的检测和一定不触发的错误目标的识别.

一方面,过程内算法迭代地计算 FAL 相关语句的格值,且算法必定在有限格高度内终止;另一方面,过程间算法基于精确的引用输入和定值输出函数副作用建模,实现了高效的过程间多项式时间、上下文敏感的 FAL 值计算.

3) 仅对可能触发的错误目标进行路径敏感的检测.

选择此类错误目标而不是传统的所有潜在错误语句作为切片种子,然后运用可能触发的错误目标的程序切片(TUSS)提取最小的路径敏感输入语句集合.

3 场景敏感检测效益分析

本文提出的场景敏感错误检测方法通过组合路径不敏感和路径敏感的检测方法达到错误检测高检测精度和高可扩展性的目标.对于静态检测而言,路径敏感的方法通常检测精度高,但可扩展性不理想;而路径不敏感的方法往往可扩展性好,但对于其有效范围外的情形检测精度不理想.我们认为,要想达到高效高精度的检测目的,不仅应考虑路径敏感部分的改进,还应考虑极大化实施路径不敏感检测的程序部分的比例,即:为了获得高检测精度,需要利用路径敏感的检测方法;而为了获得高可扩展性,则应同时考虑充分挖掘路径不敏感检测的有效范围,以相应缩小路径敏感的检测范围.

3.1 不同路径敏感检测方法的复杂度比较

设程序中最长静态路径与程序规模成正比或近正比关系,路径敏感算法分析程序 x 中一条路径的平均分析复杂度记为 $O(h(L\times V))$.其中,L 为程序 x 的规模,V 为该路径敏感分析中变量的抽象值域.在下面的讨论中,由于我们所关注的路径敏感分析的抽象值域均为变量的原值域,因此可将该复杂度简记为 $O(g(L))$.

(1) 令 $L_{original}$ 为原始程序规模,L_{sliced} 为危险语句切片后程序规模,L_{pe} 为可能触发错误目标切片后的程序规模,那么 $O(g(L_{original}))$表示危险语句切片前对于一条程序路径的平均分析复杂度,$O(g(L_{sliced}))$表示危险语句切片后对于一条路径的平均分析复杂度.

(2) 令 $N_{original\_branch}$ 为原始程序的分支数,其中,传播引擎处理的路径不敏感分析有效范围内的分支数表示为 $N1$,传播引擎分析的可能触发错误目标有关的分支数表示为 $N2$.P 表示原始程序,$P1$ 表示传播引擎处理的路径不敏感分析的有效范围,$P2$ 表示传播引擎不能有效处理的部分,有 $P2=P-P1$.所以 $N2=N_{original\_branch}-N1$.$N_{sliced\_branch}$ 表示危险语句切片后的分支数,有 $N2\leqslant N_{sliced\_branch}$.记原始程序的路径数为 $O(2^{N_{original\_branch}})$,危险语句切片后的路径数为 $O(2^{N_{sliced\_branch}})$,可能触发错误目标切片后的路径数为 $O(2^{N2})$.

基于上述表示,对于传统的全程序实施路径敏感分析的方法,其复杂度为 $O(g(L_{original})\times 2^{N_{original\_branch}})$;对于实施危险语句切片后再实施路径敏感分析的方法[3,7],其复杂度为 $O(g(L_{sliced})\times 2^{N_{sliced\_branch}})$;本文提出的针对可能触发错误实施路径敏感分析的方法,其复杂度为 $O(g(L_{sliced})\times 2^{N2})$.因为 $L_{original}\geqslant L_{sliced}$,$N_{original\_branch}\geqslant N_{sliced\_branch}$,并且 $L_{sliced}\geqslant L_{pe}$,$N_{sliced\_branch}\geqslant N2$,因此有:

$$O(g(L_{original})\times 2^{N_{original\_branch}})\geqslant O(g(L_{sliced})\times 2^{N_{sliced\_branch}})\geqslant O(g(L_{pe})\times 2^{N2}).$$

因为 $N2=N_{original\_branch}-N1$,则 $N1$ 越大,$N2$ 就越小.这说明路径不敏感分析的有效范围越大,路径敏感分析的复杂度就越小.因此,在不能降低路径敏感分析算法指数级复杂度的情况下,减少需要分析的路径数是一种使算法实用化的有效途径.

3.2 检测效益分析

由于静态检测方法存在检测精度和可扩展性两方面的要求,为了进一步刻画两者之间的关系并进一步研究静态检测方法的效益,我们基于 F-衡量[13]给出采用路径敏感分析的检测方法的检测效益模型.F-衡量用于对两个参数求加权调和平均.在本文中,F-衡量的两个参数定义如下:

假设理想的路径敏感检测精度为 A_{ideal},复杂度为 C_{ideal}.一个实际的路径敏感检测方法精度为 A,复杂度为 C.用 RA 表示实际的路径敏感检测方法的检测精度与理想的路径敏感检测精度的接近情况,令 $RA=A/A_{ideal}$,有 $RA\leqslant 1$.一般情况下,RA 值越大越好.用 RC 表示实际的路径敏感检测方法的复杂度与理想的路径敏感检测方法复杂度的接近情况,令 $RC=1-C/C_{ideal}$,$RC\leqslant 1$.一般地,在小于理想路径敏感检测方法复杂度的情况下,差距越大越好.

定义检测效益模型 $F_{Benefit}$ 如下:

$$F_{Benefit}=\frac{1}{\alpha\dfrac{1}{RA}+(1-\alpha)\dfrac{1}{RC}}=\frac{(\beta^2+1)RA\cdot RC}{\beta^2 RA+RC},$$

其中,$\beta^2=\dfrac{1-\alpha}{\alpha}$,$\alpha\in[0,1]$,$\alpha$表示 RA 和 RC 被重视的权重.假设α=1/2,那么β=1,即 RA 和 RC 同等重要,此时,

$$F_{Benefit}=\frac{2RA\times RC}{RA+RC}.$$

在检测精度相当的情况下,如果方法 1 与理想路径敏感检测方法的复杂度差距比方法 2 与理想路径敏感检测方法的复杂度的差距大的话,那么方法 1 将获得比方法 2 更大的效益.

3.3 有效性分析

在本文提出的协调路径不敏感与路径敏感分析的检测方案中,路径不敏感检测算法的部分是保守的,可以确定一定存在问题的部分.在理论上,全程序路径敏感的检测方法也会发现这部分.而对于路径不敏感方法不能确定的部分,会由路径敏感的检测方法处理.对于这部分,在理论上与全程序路径敏感检测方法的结果相同,因此,本文方法的检测精度与全程序路径敏感方法的检测精度在理论上是相当的.

一般地,考虑两种实际的路径敏感检测方法 A 和 B,计算 $\dfrac{F_{Benefit\_A}}{F_{Benefit\_B}}$,若结果大于 1,则表明方法 A 相比方法 B 能够获得更大的效益.以本文方法与传统的全程序实施路径敏感分析的方法为例,假设本文方法的检测精度为 A_{pe},复杂度为 C_{pe},那么 $RA_{pe}=A_{pe}/A_{ideal}$,$RC_{pe}=1-C_{pe}/C_{ideal}$;传统的全程序实施路径敏感分析方法的检测精度为 $A_{original}$,复杂度为 $C_{original}$,那么 $RA_{original}=A_{original}/A_{ideal}$,$RC_{original}=1-C_{original}/C_{ideal}$,有:

$$\frac{F_{Benefit\_pe}}{F_{Benefit\_original}}=\frac{\dfrac{2RA_{pe}\times RC_{pe}}{RA_{pe}+RC_{pe}}}{\dfrac{2RA_{original}\times RC_{original}}{RA_{original}+RC_{original}}} \tag{1}$$

将 RA_{pe},RC_{pe},$RA_{original}$ 及 $RC_{original}$ 代入公式(1),当 $A_{pe}\approx A_{original}$ 时,公式(1)变为

$$\frac{F_{Benefit\_pe}}{F_{Benefit\_original}}=\frac{(C_{ideal}-C_{pe})\times A_{original}\times C_{ideal}+X}{(C_{ideal}-C_{original})\times A_{pe}\times C_{ideal}+X} \tag{2}$$

其中,$X=(C_{ideal}\times C_{ideal}-C_{ideal}\times C_{original}-C_{pe}\times C_{ideal}+C_{pe}\times C_{original})\times A_{ideal}$.

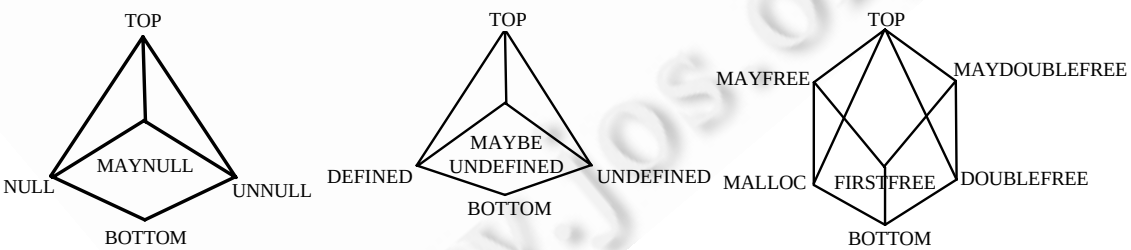
由于 $C_{pe}\leqslant C_{original}$,所以$(C_{ideal}-C_{pe})\geqslant(C_{ideal}-C_{original})$,而 $A_{original}\times C_{ideal}\approx A_{pe}\times C_{ideal}$,因此有 $\frac{F_{Benefit\_pe}}{F_{Benefit\_original}}\geqslant 1$,即 $F_{Benefit\_pe}\geqslant F_{Benefit\_original}$.该分析结果表明,本文方法与传统的全程序实施路径敏感分析的方法相比,在检测精度相当的情况下能够获得更大的效益.

4 实 现

4.1 传播引擎(PE)

结合定值-引用类错误的特征进行考虑,我们使用错误属性格建模错误属性为 FAL 值.图 6(a)给出了空指针引用错误检测的 FAL 的哈斯图[14],其中,FAL 值 UNNULL 是安全的错误属性值,NULL 和 MAYNULL 是不安全的错误属性值.且 FAL 值 NULL 具有一定不安全属性,MAYNULL 具有可能不安全属性.这样的 FAL 值分类可以帮助我们在实施错误检测的过程中有效区分一定触发和非一定触发的错误目标,其中,后者进一步划分为一定不触发和可能触发两种.

除空指针引用错误外,传播引擎还可支持其他多种具体的定值-引用类错误的检测,包括未赋值引用错误、非法指针引用错误等.对于这些错误,PE 只需根据不同的定值情形进行不同的错误属性格值设置即可.以未赋值引用错误为例,其哈斯图如图 6(b)所示,可在检测前将所有变量的错误属性格值设为表示未赋值状态的格值,如 UNDEFINED.对于存在定值的变量,其错误属性格值设为表示已定值的格值,那么当引用一个变量时,PE 将检查该变量的错误属性格值,若为表示未赋值状态的格值,即报告错误.此外,对于内存泄漏、内存两次释放、文件打开关闭等具有时序特征的错误也可有效进行错误属性格建模并实施检测.以内存泄漏和内存两次释放为例,我们在图 6(c)中给出它们的错误属性格值模型的哈斯图.在程序出口处,若变量的 FAL 值为 MALLOC,则表明存在内存泄漏;而在变量引用点,若变量的 FAL 值为 DOUBLEFREE,则表明存在内存两次释放.基于该错误属性格值模型及相应的错误属性格值计算规则,可以完成内存泄漏及内存两次释放错误的检测.



(a) 空指针引用错误 (b) 未赋值引用错误 (c) 内存泄漏及内存两次释放错误

Fig.6 FAL Hasse diagrams of typical faults

图 6 几种典型错误的 FAL 哈斯图

4.2 面向高检测精度的编译基础设施支持

为了获得高检测精度,我们在 Open64 中进行了如下扩展:

1) 基于全稀疏的 SSA[15]和域模型[16]对保守的程序分析进行了流敏感、域敏感和上下文敏感的扩展;
2) 通过构造精确的引用输入和定值输出函数副作用模型实现了准确的过程间副作用分析;
3) 内置精确、高可扩展的指针分析 LevPA[17],将流敏感、域敏感和上下文敏感的指针分析与 SSA 形式的构造相结合.LevPA 会在指针分析过程中为每个变量构造扩展的 SSA 表示,从而获得更准确和简洁

的 SSA 形式,消除虚假的引用-定值点,提高定值-引用分析的精度和效率;

4) 提供流敏感、域敏感和上下文敏感的静态程序切片支持.

5 实 验

5.1 实验环境与测试程序

我们的实验平台为 AMD Opteron 六核 CPU 服务器,主频 2.11GHz,内存 48GB.所有实验以单进程方式进行.同时,还选择 Saturn 和 Clang-sa 作为比较工具,因为它们都是有代表性的先进水平的路径敏感检测工具.同时,目前在路径敏感检测部分,我们使用 Saturn 去判定一个可能触发的错误目标是否是一个真实的错误.即:如果 Saturn 报告该可能触发的错误目标为错误,即认为它是一个真实的错误;否则,认为是 Minerva 的一个误报.在空指针检测实验中,我们直接使用 Saturn 1.2 版本提供的 null.clp 检测文件收集 Saturn 报告的空指针错误.

我们选择了 9 个 SPEC CPU2000 的 C 程序以及 3 个中大型规模的开源应用:OpenSSH[8],Apache[18]和 Wireshark[9]作为实验用例程序.这些程序应用范围广泛,包括网络应用、数据压缩算法、ICCAD 设计、游戏、字处理以及网络协议分析等.总代码规模超过了 290 万行,其中最大的单个应用超过了 200 万行.

5.2 检测结果与分析

5.2.1 不同触发场景的分类比例

Minerva 的主要创新性在于对潜在错误语句进行分类,然后对不同类别的错误目标使用不同开销的错误检测方法.因此,Minerva 的有效性在很大程度上受到所检测程序中不同类别潜在错误语句比例的影响.

对此,基于 Minerva,Clang-sa 和 Saturn 所产生的错误报告情况我们对每个实验程序中不同类别的错误目标数量进行了统计,结果如图 7 所示.统计数据来自 8 个程序,之所以没有选择全部实验程序,是因为对于另外 4 个实验程序,所有检测工具均没有找到空指针引用错误.在所观察的 8 个程序中我们发现,在 255.vortex,OpenSSH 和 Wireshark 中,一定触发和一定不触发错误目标所占比例的和超过了 90%,如此高的非可能触发错误目标比例可以使 Minerva 路径不敏感检测部分发挥很大的作用.

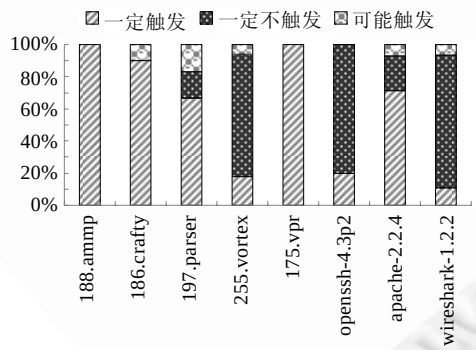


Fig.7 Different sink-triggering categories composition of 8 benchmarks

图 7 8 个测试用例中不同触发种类错误目标的组成

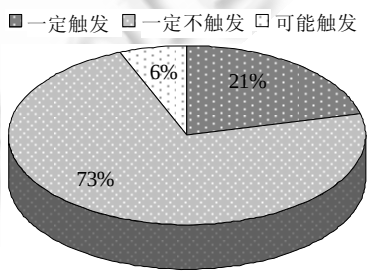


Fig.8 Percentages of sink-triggering categories across 8 benchmarks

图 8 8 个测试用例中各种错误目标的比例

我们还计算了全部 8 个实验程序中一定触发、一定不触发和可能触发的错误目标的总比例,并在图 8 中进行了显示.

对数据进行观察后不难发现,在这 8 个程序中,只有 6%的错误目标是可能触发的,其余错误目标或者是一定触发的或者是一定不触发的.一方面,这表明 Minerva 具备高可扩展性的潜质:PE 可以快速地处理那些非可能触发的错误目标;另一方面,通过识别大量的一定不触发的错误目标,PE 可以消除数量可观的误报,而这可以保证高检测精度.

5.2.2 检测时间

Minerva 的检测时间由两部分组成:一是 PE 的处理时间,包括检测一定触发和识别一定不触发的错误目标的时间,以及收集可能触发的错误目标的时间;二是对可能触发的错误目标进行路径敏感检测的时间,这部分时间由 TUSS 时间和路径敏感工具的检测时间组成.

表 2 中列举了相关实验数据,其中:P.S 列表示路径敏感的检测时间,包含 TUSS 的时间;Total 列表示 Minerva 所用的总时间.通过将 Total 和 P.S 列中的时间值相减,我们可以得到 PE 的工作时间.

Table 2 Null pointer dereference detection time comparison

表 2 空指针引用错误检测时间比较

程序	规模(KLOC)	Saturn	Clang-sa	Minerva	
				P.S	Total
ammp (188)	13.4	1 586	116	0	9
art (179)	1.2	218	12	0	1
bzip2 (256)	4.7	N/A	20	0	1
crafty (186)	21.2	903	64	98	114
equake (183)	1.5	30	6	0	1
gzip (164)	8.6	218	34	0	5
parser (197)	11.4	N/A	252	77	102
votex (255)	67.3	4 789	174	90	145
vpr (175)	17.8	774	162	0	8
openssh-4.3p2	155	3 113	377	0	57
apache-2.2.4	268.9	N/A	1 261	83	147
wireshark-1.2.2	2 383	N/A	6 282	1 035	1 935

Saturn 的检测时间通常较长,这主要是因为它需要对程序中的各条路径进行路径敏感的分析.Clang-sa 则需要探索路径可行状态,在他们的方法中,路径可行状态用于引导 Clang-sa 在实施检测时探索可行路径.为了加快检测速度,Clang-sa 仅处理一些简单的分支条件,例如,它会忽略超过一个变量的条件表达式.但即便如此,对于大规模的应用程序检测而言,Clang-sa 需要探索的空间仍会比较可观.

相对地,Minerva 的工作速度通常很快,因为大部分程序中需要进行路径敏感检测的错误目标较少;与此同时,PE 实施分类的时间也很短.举例来说,对 OpenSSH 进行空指针引用错误检测时,由于没有错误目标需要进行路径敏感的检测,所以 Minerva 仅花费了 57s 便完成了检测.

此外,我们还将 Saturn,Clang-sa 和 Minerva 检测全部 12 个程序的时间进行了比较,包括那些没有报告错误的程序.图 9 给出了比较结果.需要说明的是,在该图中,为了更好地显示时间差异很大的 3 种工具的检测时间,我们将标注时间的 *Y* 轴的最大值设为 3 000s;同时,对那些 Saturn 不能完成分析的情形,我们也将其检测时间标为 3 000s.

在检测时间的基础上,我们进一步计算了 3 种工具的检测速度,其中,Clang-sa 的平均检测速度约为每秒 338 行,Saturn 的平均检测速度约为每秒 25 行,Minerva 的平均检测速度约为每秒 1 169 行.Minerva 的平均检测速度比 Saturn 和 Clang-sa 分别快了约 46 倍和 3 倍.Minerva 在高检测速度主要得益于大比例的一定触发和一定不触发的错误目标(超过了 90%).对这些目标,Minerva 仅应用低开销的路径不敏感检测方法即完成了检测.而相对地,只有不超过 10%的可能触发的错误目标采用了路径敏感的检测方法.

5.2.3 检测精度

为了更好地进行比较,我们以空指针引用错误检测为例,将 3 种工具的检测结果在表 3 中进行了呈现.对于 Clang-sa,其检测精度上的损失原因主要是:一方面是它没有区分不同的错误触发场景,尤其是没能识别一定不触发的错误目标,产生了大量的误报;另一方面,是因为它仅对有限形式的分支条件进行处理,所以可能将不可行路径视为可行路径.Saturn 由于没有处理复杂的错误目标触发场景,同样存在与 Clang-sa 类似的误报原因;另一方面,因为没有区分复杂的场景导致需要分析的程序部分多且复杂,它也常常不能完成检测.上述两个原因使得 Saturn 的检测精度低于 Minerva.

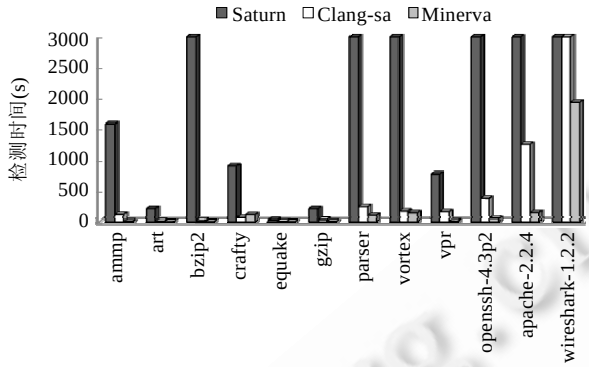


Fig.9 Detection time of Saturn, Clang-sa and Minerva

图 9 Saturn,Clang-sa 和 Minerva 的检测时间

Table 3 Null pointer dereference detection results comparison

表 3 空指针引用错误检测结果比较

程序	Saturn			Clang-sa			Minerva		
	TF	FP	RFN	TF	FP	RFN	TF	FP	RFN
ammp (188)	0	4	6	5	0	1	6	0	0
art (179)	0	0	0	0	0	0	0	0	0
bzip2 (256)	N/A	N/A	N/A	0	0	0	0	0	0
crafty (186)	5	0	4	0	0	9	9	1	0
equake (183)	0	0	0	0	0	0	0	0	0
gzip (164)	0	0	0	0	0	0	0	0	0
parser (197)	N/A	N/A	N/A	2	2	2	4	1	0
votex (255)	5	15	4	9	41	0	9	3	0
vpr (175)	1	1	0	0	2	1	1	0	0
openssh-4.3p2	1	4	1	2	8	0	2	0	0
apache-2.2.4	N/A	N/A	N/A	10	3	0	10	1	0
wireshark-1.2.2	N/A	N/A	N/A	15	178	8	23	14	0

此外,我们还计算了所有工具的平均误报率和相对漏报率,结果参见表 4.其中,相对漏报主要是指相对于其他两个工具统计一个工具的漏报数.对于本文所用的实验用例,Saturn 和 Clang-sa 的误报率都表现较高,特别是 Clang-sa,其误报率高达 84%,这主要是因为实验用例中存在大量的 SCF 而 Clang-sa 不能识别.

Table 4 Average false positive and relative false negative rate

表 4 平均误报率和相对漏报率

	Saturn (%)	Clang-sa (%)	Minerva (%)
误报率	67	84	24
相对漏报率	56	38	0

Saturn 和 Clang-sa 的相对漏报率也不理想,这主要表现为它们的过程间分析尚不完善,如未能识别全局量的相关错误,并且它们也未能识别特殊库函数的不安全返回值,如内存分配、字符串处理等可能返回 NULL 的函数,从而导致了漏报.比如在 186.crafty 中,Saturn 和 Clang-sa 的漏报主要就是因为这个原因.

综上,对于本文所用的实验用例,Minerva 的平均误报率为 24%,约为 Saturn 和 Clang-sa 的 1/3,且没有漏报已知错误.这样的表现主要与下列因素相关:1) 通过识别一定不触发的错误目标,消除了大量潜在的误报;2) 精确的程序分析基础设施支持;3) 精确的过程间副作用建模.

6 相关工作

(1) 路径敏感的错误检测

近年来,出现了不少路径敏感的错误检测研究工作.Xie 等人[19]提出了 ARCHER,它是一个符号的、路径敏感的检测内存错误的工具.ARCHER 对每个内存访问点的访问约束进行分析,因此需要对每条可能的执行路径进行穷尽分析.Aiken 等人[20,21]将一个程序基于可满足性求解框架进行建模,然后实施检测.他们的模型是精确

的,但遭遇大规模程序时,条件的表示和约束求解会变得非常复杂.Das 等人[4]在程序中对分支的属性相关行为进行建模,以避免完全路径敏感分析潜在的指数级开销.他们在控制流汇合点处对相同的属性状态进行合并,这意味着若属性状态不同,他们仍需对每条路径进行路径敏感的分析.Kremenek 等人[2]利用编译器框架去寻找典型的程序错误,并在不同的执行路径上探索可行符号执行状态.所有上述这些方法,因为使用了路径敏感,在精度上都有一定的保证,然而可扩展性并不理想.与他们不同的是,Minerva 从扩大路径不敏感检测有效范围的角度出发,通过对不同的代码区域使用不同开销的检测方法,避免了统一路径敏感方法存在的大量冗余开销,不仅获得了高可扩展性,还保证了高检测精度.

(2) 需求驱动的错误检测

为了处理路径敏感分析的大开销问题,需求驱动的策略常被采用.Nanda 等人[3]执行后向需求驱动分析去识别有不安全值流向错误引用点的路径,以避免穷尽的程序空间探索.但是,由于需要在每条解引用语句上去执行后向的分析,他们的方法需要分析的路径数仍然很大.Parfait[5]是 Sun 开发的一个静态的、分层的程序分析检测框架,他们利用一组不同的程序分析实施错误检测.为了控制开销,Parfait 选择低开销的简单的程序分析解析那些容易检测的错误,而对那些复杂错误才应用开销较大的程序分析.以缓冲区溢出错误检测为例,他们分别应用常量传播与折叠、部分计算以及关联约束的符号分析去完成检测.Le 等人[6]提出了一种借助需求驱动的路径敏感分析提炼缓冲区溢出错误检测的方法和框架——Marple[7],其中,需求被建模为一组对兴趣语句的查询,错误的检测基于提出、传播、更新以及求解查询完成.上述方法都使用了需求驱动的路径敏感方法去减少总开销,但他们仅排除了与错误无关的路径,仍留有大量与错误有关的路径需要分析.而 Minerva 通过不同错误触发类别的划分,首先排除错误相关路径中一定不触发的路径,然后仅对可能触发的错误目标实施路径敏感的检测,并在检测前采用可能触发错误目标制导的切片策略,进一步缩小路径敏感检测部分的输入.例如,对于我们的实验用例,传统的以潜在错误语句为种子的切片方法去除了约 82%的路径数,而基于可能触发错误制导的切片方法则去除了约 98%的路径数.

(3) 基于值流分析的检测

有一些方法使用值流分析去检测错误.Cherem 等人[22]提出了 Fastcheck,一个基于值流分析的错误检测工具.它为整个程序构造一个值流图,并在该图上执行部分路径敏感的检测.Sui 等人[15]基于全稀疏值流分析实现了一个内存泄漏检查器 SABER.他们为程序构造稀疏的值流图(SVFG),然后,基于该图去执行内存泄漏检查.与上述方法不同的是,Minerva 是沿着定值-引用链去追踪 FAL 值以寻找危险的错误目标.

将静态分析应用于错误检测,是程序语言和编译技术以及软件工程领域的一个重要研究方向[23].围绕静态检测精度与效率的目标,张阳等人[24]在传统源代码安全属性验证工具基础上加入指针逻辑,利用指针逻辑对源码的分析结果,对源代码中的指针进行替换,加强了静态代码属性验证工具的指针处理功能.陈意云等人[25]改进并扩展了为验证指针程序提出的指针逻辑,提出合法访问路径集合的概念,使指针逻辑能够更方便地应用于函数调用.肖庆等人[26]采用抽象取值范围表示变量的取值信息,计算每个程序位置上状态机的可能属性状态集合,用变量抽象取值范围为空表示不可达路径.赵云山等人[27]利用基于缺陷的程序切片方法去除缺陷无关节点,减少路径敏感分析方法的误报,同时提高缺陷检测效率.崔展齐等人[28]提出一种结合静态分析和混合执行测试技术的目标制导的混合执行测试方法,将静态分析检测与测试技术相结合,发现程序中的缺陷.此外,形式化方法作为计算机系统及软件验证的重要途径之一,也为高可信软件提供了重要支持[29].

7 结论与展望

本文提出一种高效的场景敏感的定值-引用类错误检测方法,该方法基于对潜在错误语句的分类进行检测.本文方法充分考虑不同的错误目标触发场景,然后基于不同的错误目标触发情形,组合具有合适精度和开销的不同检测方法,达到高可扩展性和高精度的检测目标.本文为上述方法实现了一个原型系统 Minerva,并将其运用到广泛应用的、总代码规模超过 290 万行的实际程序中.通过以空指针引用错误检测为实例研究,本文的实验用例结果表明,Minerva 的检测时间比已有的先进水平的检测工具 Clang-sa 和 Saturn 平均快了约 3 倍和 46

倍,且误报率仅为 24%,约为 Clang-sa 和 Saturn 误报率的 1/3,并且没有发现漏报已知错误.所有这些数据表明,Minerva 的场景敏感的检测方法是一种高效、准确、实用的错误检测方法.

References:

[1] Larson E. A plethora of paths. In: Proc. of the 17th IEEE Int'l Conf. on Program Comprehension. IEEE Press, 2009. 40−49.

[2] Kremenek T. Finding software bugs with the clang static anayzer. Apple Inc., 2008. http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

[3] Nanda MG, Sinha S. Accurate interprocedural null-dereference analysis for Java. In: Proc. of the 31st Int'l Conf. on Software Engineering. ACM Press, 2009. 133−143.

[4] Das M, Lerner S, Seigle M. ESP: Path-Sensitive program verification in poly-nomial time. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM Press, 2002. 57−58. [doi: 10.1145/512529.512538]

[5] Cifuentes C, Scholz B. Parfait: Designing a scalable bug checker. In: Proc. of the 2008 Workshop on Static Analysis. ACM Press, 2008. 4−11. [doi: 10.1145/1394504.1394505]

[6] Le W, Soffa ML. Refining buffer overflow detection via demand-driven path-sensitive analysis. In: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM Press, 2007. 63−68. [doi: 10.1145/1251535.1251546]

[7] Le W, Soffa ML. Marple: A demand-driven path-sensitive buffer overflow detector. In: Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2008. 272−282. [doi: 10.1145/1453101.1453137]

[8] OpenSSH. http://www.openssh.org/

[9] Wireshark. http://www.wireshark.org/

[10] Open64. http://www.open64.net/

[11] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Techniques and Tools. 2nd ed., Amazon Press, 2007. 618−630.

[12] Ngo MN, Tan HBK. Detecting large number of infeasible paths through recognizing their patterns. In: Proc. of the 15th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2007. 215−224. [doi: 10.1145/1287624.1287655]

[13] Manning CD, Raghavan P, Schutze H. Introduction to Information Retrieval. Cambridge University Press, 2008. 359−360.

[14] Pemmaraju S, Skiena S. Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Cambridge University Press, 2003. 352−362.

[15] Sui YL, Ye D, Xue JL. Static memory leak detection using full-sparse value-flow analysis. In: Proc. of the Int'l Symp. on Software Testing and Analysis. ACM Press, 2012. 254−264. [doi: 10.1145/04000800.2336784]

[16] Yu HT, Zhang ZQ. An aggressively field-sensitive unification-based pointer analysis. Chinese Journal of Computers, 2009,32(9): 1722−1735 (in Chinese with English abstract).

[17] Yu HT, Xue JL, Huo W, Feng XB, Zhang ZQ,. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proc. of the 8th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. ACM Press, 2010. 218−229. [doi: 10.1145/1772954.1772985]

[18] Apache. http://www.apache.org/

[19] Xie YC, Chou A, Engler D. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In: Proc. of the 11th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2003. 327−336. [doi: 10.1145/940071.940115]

[20] Aiken A, Bugrara S, Dillig I, Dillig T, Hackett B, Hawkins P. An overview of the Saturn project. In: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM Press, 2007. 43−48. [doi: 10.1145/1251535.1251543]

[21] Dillig I, Dilig T, Aiken A. Sound, complete and scalable path-sensitive analysis. In: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM Press, 2008. 270−280. [doi: 10.1145/1379022.1375615]

[22] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM Press, 2007. 480−491. [doi: 10.1145/1250734.1250789]

[23] Zhang J. Sharp static analysis of programs. Chinese Journal of Computers, 2008,31(9):1549−1552 (in Chinese with English abstract).

[24] Zhang Y, Cheng L. A new property verification method for code security based on pointer logic. Chinese Journal of Computers, 2009,32(6):1119−1125 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01119]

[25] Chen YY, Li ZP, Wang ZF, Hua BJ. Pointer logic for verification of pointer programs. Ruan Jian Xue Bao/Journal of Software, 2010,21(3):415−426 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3620.htm [doi: 10.3724/SP.J.1001.2010.03620]

[26] Xiao Q, Gong YZ, Yang ZH, Jin DH, Wang YW. Path sensitive static defect detecting method. Ruan Jian Xue Bao/Journal of Software, 2010,21(2):209−217 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3782.htm [doi: 10.3724/SP.J.1001.2010.03782]

[27] Zhao YS, Gong YZ, Li L, Xiao Q, Yang ZH. Improving the efficiency and accuracy of path-sensitive defect detecting. Chinese Journal of Computers, 2011,34(6): 1100−1113 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01100]

[28] Cui ZQ, Wang LZ, Li XD. Target-directed concolic testing. Chinese Journal of Computers, 2011,34(6):953−964 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.00953]

[29] Wang J, Li XD. Preface to special issue on formal methods and tools. Ruan Jian Xue Bao/Journal of Software, 2011,22(6):1121−1122 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4036.htm [doi: 10.3724/SP.J.1001.2011.04036]

附中文参考文献:

[16] 于洪涛,张兆庆.激进域敏感基于合并的指针分析.计算机学报,2009,32(9):1722−1735.

[23] 张健.精确的程序静态分析.计算机学报,2008,31(9):1549−1552.

[24] 张阳,程亮.一种基于指针逻辑的代码安全属性分析方法.计算机学报,2009,32(6):1119−1125. [doi: 10.3724/SP.J.1016.2009.01119]

[25] 陈意云,李兆鹏,王志芳,华保健.一种用于指针程序验证的指针逻辑.软件学报,2010,21(3):415−426. http://www.jos.org.cn/1000-9825/3620.htm [doi: 10.3724/SP.J.1001.2010.03620]

[26] 肖庆,宫云战,杨朝红,金大海,王雅文.一种路径敏感的静态缺陷检测方法.软件学报,2010,21(2):209−217. http://www.jos.org.cn/1000-9825/3782.htm [doi: 10.3724/SP.J.1001.2010.03782]

[27] 赵云山,宫云战,刘莉,肖庆,杨朝红.提高路径敏感缺陷检测方法的效率及精度研究.计算机学报,2011,34(6):1100−1113. [doi: 10.3724/SP.J.1016.2011.01100]

[28] 崔展齐,王林章,李宣东.一种目标制导的混合执行测试方法.计算机学报,2011,34(6):953−964. [doi: 10.3724/SP.J.1016.2011.00953]

[29] 王戟,李宣东.形式化方法与工具专刊前言.软件学报,2011,22(6):1121−1122. http://www.jos.org.cn/1000-9825/4036.htm [doi: 10.3724/SP.J.1001.2011.04036]

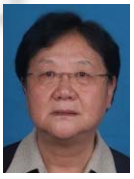
衷璐洁(1979－),女,江西南昌人,博士生,讲师,主要研究领域为程序分析,错误检测.
E-mail: zhonglujie@ict.ac.cn

李丰(1985－),女,博士,CCF 会员,主要研究领域为程序分析,错误调试.
E-mail: lifeng2005@ict.ac.cn

霍玮(1981－),男,博士,助理研究员,CCF 会员,主要研究领域为程序分析,错误检测.
E-mail: huowei@iie.ac.cn

冯晓兵(1969－),男,博士,研究员,博士生导师, CCF 高级会员,主要研究领域为先进编译技术及相关工具环境.
E-mail: fxb@ict.ac.cn

李龙(1988－),男,工程师,主要研究领域为程序分析,错误检测.
E-mail: lilong@ict.ac.cn



张兆庆(1938－),女,研究员,博士生导师, CCF 高级会员,主要研究领域为编译技术及相关工具.
E-mail: zqzhang@ict.ac.cn