第41卷 第3期 计 算 机 学 报 Vol. 41 No. 3
2018年3月 CHINESE JOURNAL OF COMPUTERS Mar. 2018

面向二进制程序的空指针解引用错误的检测方法

傅 玉[1)] 邓 艺[1)] 孙晓山[1)] 程 亮[1)] 张 阳[1)] 冯登国[1),2)]

[1)](中国科学院软件研究所可信计算与信息保障实验室 北京 100190)

[2)](中国科学院大学 北京 100190)

摘 要 空指针解引用是C/C++程序中常见的一类程序错误,它可让攻击者旁路安全机制或窥探操作系统敏感信息,一直是计算机安全领域的重要研究课题之一.目前已有很多(自动)分析工具对其进行检测,然而它们都在源代码层面上进行检测.大量的商业软件不公开源代码,因此基于源代码的工具无法对这类软件中空指针解引用进行检测.此外,一些空指针解引用无法在源代码层面检测,因为这些缺陷由编译选项和编译优化不当引入.因此进行基于二进制的空指针解引用检测非常必要.基于二进制的空指针解引用检测的一个优势是可以包含库函数的代码,而基于源代码的分析通常采用人工构造的库函数摘要,从而影响检测的准确性和召回率.该文首次提出并实现了面向二进制程序的空指针解引用静态检测工具NPtrChecker,直接接受二进制程序进行分析,并给出代码中出现空指针的来源和解引用的位置以及对应的路径条件.在二进制上进行空指针解引用检测的一个重要难点是二进制程序中缺少指针类型、结构体类型等相关数据类型信息.如果缺乏这类信息,会导致分析结果的准确率大大降低.但是从二进制中恢复类型、数据结构本身是非常困难的问题.针对上述问题,我们提出了一种内存模型,区分来自同一数据结构的不同域的引用,实现了针对空指针解引用检测的域敏感指针分析.为了进一步提高分析的准确率,文章在此基础上设计实现了一套基于函数摘要的上下文敏感的数据流分析算法.此外,工具采用最弱前置条件对数据流分析结果进行验证,检查从指针来源到解引用点的路径条件是否可以被满足,以降低误报率.我们应用NPtrChecker分析了SPEC2000中的11个程序,总共报告了37个可疑空指针解引用,通过人工确认,其中22个是真实的程序错误.相对于Saturn报告的92个,仅13个为真;LUKE报告的3个,2个为真,而文中的工具检测出了更多的空指针解引用错误,同时保持了较低的误报率.

关键词 空指针解引用检测;静态程序分析;二进制程序分析;最弱前置条件;数据流分析

中图法分类号 TP309 **DOI号** 10.11897/SP.J.1016.2018.00574

计算机学报

Finding Null Pointer Dereference in Binaries

FU Yu[1)] DENG Yi[1)] SUN Xiao-Shan[1)] CHENG Liang[1)] ZHANG Yang[1)] FENG Deng-Guo[1),2)]

[1)](*Trusted Computing and Information Assurance Laboratory, Institute of Software Chinese Academy of Sciences, Beijing* 100190)

[2)](*University of Chinese Academy of Sciences, Beijing* 100190)

Abstract Null pointer dereference, which may allow attackers to bypass security logic or reveal sensitive information in operating system, is a common programming bug in C/C++ programs and therefore has been an important research subject in computer security. Many (automatic) analysis tools have been proposed to detect this bug. However, all these tools run on the source code. Many commercial softwares come with no source code, and these tools running on the source code cannot detect null pointer dereferences in them. Some null pointer dereference defects are introduced by the compiling configurations or the compiler's optimizations and the tools running on the source code cannot detect them as well. So it is necessary to develop tools that can detect null pointer dereference directly on the binaries. One advantage of null pointer dereference

收稿日期:2016-06-28;在线出版日期:2017-04-19.本课题得到国家自然科学基金(61471344)、国家242信息安全计划(2016A086)资助.
傅 玉,男,1988年生,博士研究生,主要研究方向为程序分析、系统安全.E-mail:fuyu_iscas@outlook.com.**邓 艺**,男,1980年生,博士,助理研究员,主要研究方向为程序分析、系统安全.**孙晓山**,男,1979年生,博士,助理研究员,主要研究方向为程序分析、系统安全.**程 亮**,男,1982年生,博士,副研究员,主要研究方向为程序分析、系统安全.**张 阳**,女,1971年生,博士,副研究员,主要研究方向为系统安全.**冯登国**,男,1965年生,博士,研究员,主要研究领域为信息安全、密码学.

detection on the binaries is that the library code will be included in the analysis and the source code based detections usually use crafted summaries for the library function which may lower the precision and recall. In this paper, we present and implement NPtrChecker, the first analysis tool for null pointer dereferences detection, which accepts binary programs as inputs, outputs the location where null pointers come from, where dereferences occur and the corresponding path conditions. One difficulty of null pointer dereferences detection on the binaries is that the information about pointer type and structure type is missing in binaries. Without these information, the detection can have bad performance on accuracy. However, it is hard to recover the data structure definitions from the binaries. We propose a memory model to differentiate different fields of data structures without restoring the data structures and type information. This memory model is the basis of our field sensitive pointer analysis for null pointer dereference analysis. We continue to design a context sensitive dataflow analysis algorithm based on the function summary techniques, and the algorithm improves the precision of the analysis. To reduce false positives as many as possible, we also leverage weakest precondition to filter out the unreachable paths reported by dataflow analysis. The report will be removed if the path conditions from source to the sink that dereferences the pointer cannot be satisfied. We apply NPtrChecker to 11 programs in the SPEC2000 benchmark and 37 suspicious null pointer dereference defects are reported. Among them 22 reports are proved to be true defects by manual examinations. In contrast, the tool Saturn reports 92 defects and only 13 are true positive and LUKE reports 3 defects and only 2 are true positive. This shows that our method can detects more null pointer dereferences and keep the false positive at low level.

Keywords　static program analysis; null pointer detection; binary analysis; weakest precondition; dataflow analysis

1　引　言

随着信息化技术的发展,人们对软件可靠性提出了越来越高的要求.空指针解引用是计算机程序中的一种常见错误,通常会引起程序崩溃,进而导致程序的拒绝服务或系统安全机制的失效.如图 1 所示中国国家信息安全漏洞库提供①的数据表明:在软件厂商安全意识逐年提高情况下,近年来空指针解引用漏洞仍然保持着较高的数量.空指针漏洞广泛分布于操作系统(Linux 内核、Mac OS、iOS)、数据库(Firebird、MySQL)、浏览器(IE、Firefox、Safari)、办公软件(Microsoft Office)、网络协议(OpenSSL)、网页服务器(Apache HTTP Server)等软件中.发现并排除程序中的空指针解引用错误可以提高程序可靠性,减少安全威胁,具有重要意义.

空指针解引用的检测,可以抽象为判定程序中是否存在一条从函数的入口点开始,经过空指针产生点,并最终解引用该指针的可达路径.工业界和学术界近年来提出了多种检测方案,Saturn[1-2],Archer[3],Calysto[4]等工具采用符号执行来收集路径条件,并调用约束求解器验证路径的可达性.XYLEM[5],文献[6]、DTS[7]将路径问题转换为最弱前置条件进行求解.FindBugs[8-9],文献[10-11]利用类型一致性与模式匹配来选择最有可能的空指针解引用.LUKE[12]通过建立保障值依赖图,只分析与指针相关的路径

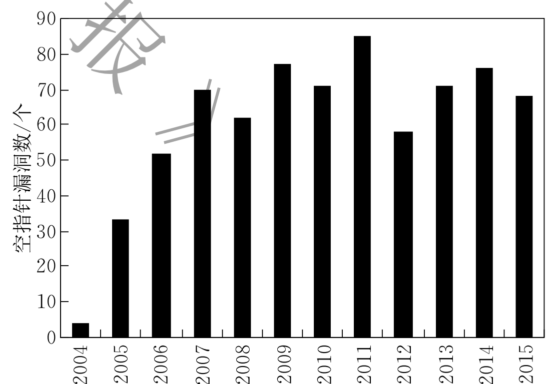

图 1　空指针解引用漏洞分布图(2004～2015)

① http://www.cnnvd.org.cn/vulnerability/statistics

《计算机学报》

条件,进而提高验证的速度.

然而以上现有研究都是针对程序的源代码进行检测,而使用源代码作为检测对象具有一定的局限性,从而限制了工具的可用性.首先,源代码在一些情况下不能获得.例如:部分商业软件出于保护自己利益的考虑,不会对外公布源代码;一些第三方函数库只以二进制文件的形式提供给用户.其次,编译过程会影响源代码的语义:一些错误是由源代码编译为二进制代码环节由编译器引入的[13],面向源代码的方法无法检测出这类漏洞.因此研究面向二进制程序中的空指针错误检测方法很有意义.

相对于Saturn、LUKE这类面向源代码的检测方法,面向二进制程序的空指针错误检测面临以下几个挑战.首先,二进制指令类型繁多,语义复杂,直接进行分析的难度大.其次,相对于源代码,二进制代码中缺少变量名称和指针、结构体类型等高层语义信息,加大了指针分析的难度.这些缺陷使的在二进制程序上进行空指针检测具有很大的难度.

为了解决在二进制分析上的困难,特别是解决现有空指针解引用错误检测方案的局限性,我们进行了如下处理.首先,针对二进制代码指令复杂难于分析的特点,我们引入一种中间语言作为程序的表示方法,避免直接在二进制代码上进行分析.其次,对于二进制代码高级语义信息缺失的问题,我们设计了一个层次化的内存模型,用于表示指针变量的关系.然后在此内存模型上,设计了一套基于函数摘要的上下文敏感、域敏感的空指针数据流分析算法.该算法在不损失精度的前提下,尽可能的提高了效率.最后,为进一步降低误报率,我们使用最弱前置条件对数据流分析的结果进行验证.

我们设计并实现了检测工具NPtrChecker(Null Pointer Checker),用于检测C/C++的二进制程序中的空指针解引用错误,目前支持分析Windows平台上的X86二进制可执行文件.NPtrChecker输出空指针的产生地址、解引用地址、函数调用序列等相关信息,帮助安全分析人员和软件开发者完成对程序错误的分析.

我们应用NPtrChecker分析了SPEC2000中的11个程序,总共报告了37个可疑空指针解引用,通过人工确认,其中22个是真实的程序错误.相对于Saturn报告的92个,仅13个为真;LUKE报告的3个,2个为真,我们的工具检测出了更多的空指针解引用错误,同时保持了较低的误报率.

综上所述,本文的主要贡献如下:

(1)设计并实现了首个针对二进制程序空指针解引用错误检测方案.该方案利用流数据流分析和函数摘要的方法,筛选出疑似的空指针解引用,并利用最弱前置条件,对这些错误进行验证.

(2)为了处理二进制程序中的内存访问,设计了一套内存模型,实现了域敏感和上下文敏感的指针分析算法.

(3)基于以上技术,实现了原型系统NPtrChecker.我们使用该工具对SPEC2000中的程序进行检测,与已有的开源工具Saturn和LUKE进行相比,我们的工具发现了更多的空指针解引用错误,同时保持了较低的误报率.

本文第2节介绍相关工作及研究动机;第3节为本文工具的总体结构以及工作流程示例;第4节详细介绍工具的各个模块以及为有效检测漏洞所采用的各项技术;第5节测试NPtrChecker的有效性和检测效率,并将实验结果与现有工具进行了对比分析;最后在第6节总结我们的工作,并指出未来的研究方向.

2 研究动机

2.1 相关工作

空指针解引用作为一种常见的程序错误,一直是计算机安全领域的重要研究问题之一.下面对其中一些具有代表性的检测方法和工具进行介绍.

Saturn[1-2]是面向C源程序的静态检测工具,它将空指针检测问题转换为SAT求解问题.Saturn在函数内进行路径敏感的分析,为了避免路径爆炸,在过程间使用函数摘要完成分析.Saturn实现了上下文敏感、域敏感的空指针检测算法,但是不准确的函数摘要和激进的检测策略导致了大量的误报.Calysto[4]采取了与Saturn类似的思路,使用最大共享图来优化路径条件的表达,提高约束求解的效率,实现了过程间的路径敏感分析.

LUKE[12]使用保障值依赖图确定与空指针解引用相关的变量,只收集这些变量的约束条件,减少了约束求解时的计算量,提高了检测效率.LUKE还对库函数进行了更好的建模,增加了被检测到的空指针错误的数量.LUKE不能识别出不同源文件间的函数调用关系,很多程序路径没有分析到.

FindBugs[8-9]是面向java字节码的检测工具,通过人工经验总结空指针问题的常见原因,构造缺

《计算机学报》

陷模式，检测过程内的空指针错误. FindBugs 是一个路径不敏感的分析算法，对于复杂路径上的空指针解引用，FindBugs 直接认为这类空指针不可达，这种过于粗犷的方式，在对实际程序的检测中造成了漏报的情况.

XYLEM[5] 从指针的解引用点开始，进行反向的控制流分析，在分析过程中传播并收集路径约束条件，直至分析到函数入口点，或者约束条件无法被满足. 这种需求驱动的方法，避免了对程序路径的穷举遍历，适用于大规模的程序. 但是，XYLEM 并没有实际降低分析的工作量，只是将一次正向分析，变成了多次逆向分析.

2.2　研究动机

空指针解引用是程序中常见的一类错误，从单个函数的角度来看，空指针来源于以下几个途径：

(1) 程序语句的直接赋值. 例如编程者在声明一个指针类型变量时，将 NULL 作为该指针的初始值.

(2) 危险的库函数. 例如 malloc、realloc 等涉及内存分配的库函数；strchr、strtok 等用于字符串操作的库函数.

(3) 函数的输入变量. 包括该函数的参数、访问的全局变量，以及这些变量的域.

(4) 调用其他函数的影响. 其他函数中的空指针可能通过函数返回值、参数和全局变量传递给当前函数.

前两种空指针可通过过程内分析和对危险库函数进行特别处理来获得准确的信息，后两者则需要过程间、上下文敏感、域敏感的分析才能获得准确的信息.

此外，空指针解引用的判定，通常需要考虑路径条件，如下列代码：

```
01  int pfun(int level) {
02    int* x=null;
03    int i;
04    if (level>0) then x=&i;
05    if (level>4) then return  *x;
06    return 0;}
```

假如不考虑路径条件，通过数据流分析可知，第 5 行指针 x 可能来自第 2 行的赋值，也可能来自第 4 行的赋值. 这是数据流分析所能获得的全部信息，此时可以为了发现更多的错误，报告这个错误；或者为了减少误报，不报告这个错误.

(1) 假如报告这个错误. 通过分析路径条件，要使程序在第 5 行解引用 x，level 必须大于 4. 而 level 大于 4，则第 4 行必然执行，使 x 指向 &i，故最终解引用时不会出现空指针解引用. 这是一个误报.

(2) 假如不报告这个错误. 对于这个例子，这样做是正确的. 但是如果第 5 行的条件变为 if (level<4) 时，不报告就会产生了一个漏报.

从上面这段代码可以看出，为了降低误报率和漏报率，对代码路径条件进行分析是不可或缺的.

3　系统概述

3.1　NPtrChecker 系统概述

为了解决上述空指针检测过程中可能遇到的问题，我们设计并实现了检测工具 NPtrChecker，该工具实现了域敏感、上下文敏感的过程间数据流分析，并使用最弱前置条件对数据流分析结果进行验证，降低了误报率. 如图 2 所示，NPtrChecker 由预处理模块、空指针检测模块、空指针验证模块三部分组成.

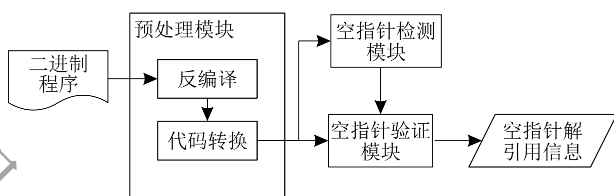

图 2　NPtrChecker 架构图

在预处理模块中，我们使用 IDA Pro 反汇编待检测的二进制程序，获得对应的汇编指令、函数划分、函数调用图等重要信息. 由于汇编指令语义的复杂性，我们将汇编指令转化为 Vine[14] 中间语言，使用静态单一赋值(Static Single-Assignment，SSA)来表示中间语言中的变量，方便后续的分析.

空指针检测模块负责找出可疑的空指针解引用. SSA 无法有效处理内存上的读写操作，故需要在中间语言上进行指针分析. 受底层语言指针分析算法 VLLPA(VELOCITY Low-Level Pointer Analysis)[15] 的启发，我们使用多层偏移地址来表示函数中访问过的内存地址，例如：参数 p 来自栈偏移地址 p_off，则 $*p$ 和 $*(p+4)$ 对应的多层偏移地址为 p_off@0 和 p_off@4. 这种方法有效的描述了变量之间的域关系，并为内存地址提供了直观的表达形式. 我们扩展了 VLLPA 算法，将空值信息的分析加入了指针分析中，满足了实验需求.

我们在单个函数内部识别并标记访问过的所有内存地址，建立变量与内存地址之间的关系. 以此为基础，我们在函数内部进行数据流分析，并利用函数

《计算机学报》

摘要，实现上下文敏感、域敏感的过程间分析.

空指针验证模块负责验证可疑空指针解引用的有效性.对每个可疑的空指针解引用，我们根据检测模块中的数据流分析所提供的指针产生点和解引用点地址以及预处理中获得的函数调用关系，计算出从空指针产生点到空指针解引用点的函数调用序列，并采用函数内联的方式将相关函数放入同一个程序控制流图中.然后通过反向分析，收集从解引用点开始，反向经过空指针产生点，最后到函数入口的最弱前置条件，调用约束求解器对收集到的条件信息进行求解.若有解，则表明从空指针产生点到解引用点的路径是可达的，于是认为这是一个有效的空指针解引用.

3.2 空指针检测示例

我们通过图3所示的空指针解引用示例来介绍检测工具 NPtrChecker 的工作流程.图3(a)给出了待检测函数 fun1 和 fun2 的源码，以及它们使用的结构体 st.图3(b)给出了两个函数对应的关键汇编代码.

```
01 typedef struct{int i; int** j;} st;
02 void fun1 (st *p){
03     *(p->j)=NULL; }
04 void fun2() {
05     int n=1;
06     int *np=&n;
07     st sp={n, &np};
08     fun1(&sp);
09     *np=2;
10     }
```

(a)

```
fun1:
…
0x4113be mov eax, [ebp+arg_0]
0x4113c1 mov ecx, [eax+4]
0x4113c4 mov dword ptr [ecx], 0
…
fun2:
…
0x4113fe mov [ebp+var_8], 1
0x411405 lea eax, [ebp+var_8]
0x411408 mov [ebp+var_14], eax
0x41140b mov eax, [ebp+var_8]
0x41140e mov [ebp+var_24], eax
0x411411 lea eax, [ebp+var_14]
0x411414 mov [ebp+var_20], eax
0x411417 lea eax, [ebp+var_24]
0x41141a push eax
0x41141b call fun1
0x411420 add esp, 4
0x411423 mov eax, [ebp+var_14]
0x411426 mov dword ptr [eax], 2
…
```

(b)

图3 空指针解引用数据流分析示例

在图3(a)的第7行，fun2 实例化了结构体 st 的对象 sp，并将 $sp.j$ 初始化为 $\&np$，np 是一个指针.第8行调用了 fun2，fun2 执行后将 $*(st.j)$ 修改为空，$st.j$ 为 $\&np$ 的别名指针，故 np 也被修改为空，进而直接导致第9行解引用指针 np 时产生了一个空指针解引用错误.NPtrChecker 采用自底向上的数据流分析，先分析被调函数 fun1，生成它的函数摘要；然后将 fun1 的函数摘要用于主调函数 fun2 的分析中.

对任意待分析的函数，我们认为该函数中被访问的参数、全局变量以及它们域上的值，都是潜在的空指针.因此对 fun1 完成数据流分析后，NPtrChecker 会报告两处可能的空指针解引用，分别是指令 0x4113c1 的 $eax+4$ 以及 0x4113c4 的 ecx，对应源代码中的指针 p 和 $p->j$.假设 fun1 的初始栈偏移为 $sf1$，p 所在地址为 $sf1+4$.由于指针 p 的值未知，我们将 $p->j$ 的地址表示为多层偏移地址形式 $(st1+4)@4$，这个表达式的含义是：$p->j$ 的地址为指针 p 所指向地址位置加上4(p 指向的地址是 $p->i$，故 $p->j$ 的地址还要再加4).

NPtrChecker 的数据流分析除了记录可能的空指针解引用之外，还会分析函数对外部数据的影响，影响范围包括函数参数、全局变量以及它们的域和函数的返回值.在 fun1 中，指令 0x4113C4 处内存[ecx]被修改为0，即源代码中的 $*(p->j)=$ NULL.在这个过程中，多层偏移地址 $(st1+4)@4@0$ 上的内存对象被修改为 NULL.

函数摘要包含可疑空指针解引用集合和内存修改集合这两部分.当分析到 fun2 调用 fun1 时，NPtrChecker 使用 fun1 的函数摘要.在此上下文环境中，fun1 中两个可能导致空指针解引用的指针都来自 fun2，NPtrChecker 试图找出 fun2 中对应的指针是哪个，以判断是不是可疑空指针解引用.fun2 调用 fun1 时对应的内存分布如图4所示，第一个可疑空指针 p 的偏移为 $(sf1+4)$，对应 fun2 中的偏移 $(sf2-72)$，该内存地址上储存的值是 $\&st$，$\&st$ 不为空.所以 fun1 中被解引用的指针 p 在当前上下文环境下并不是空指针，不会导致空指针解引用.fun1 中的第二个可疑指针 $p->j$，它的多层偏移地址为 $(st1+4)@4$，NPtrChecker 首先查找 $(st1+4)$ 上的指针对应着 fun2 中的哪个指针，通过分析发现是 $\&(st)$ 这个指针，之后对 $\&(st)$ 进行解引用，获得的值再加上偏移量4，便找到了对应的地址 $sf2-32$，储存在这个偏移上的指针为 $st.j$，不为 NULL，fun2 中的第二个可疑指针也不为空，不会导致空指针解引用错误.

在处理 fun1 函数摘要的可疑空指针解引用集合后，NPtrChecker 继续处理函数摘要中的内存修

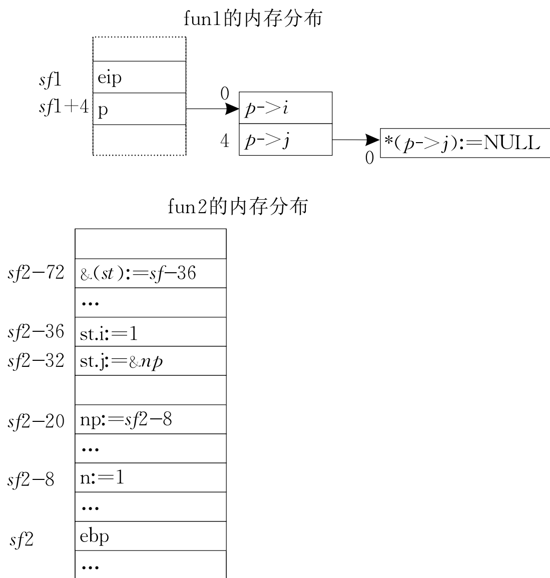

图 4　fun1、fun2 的内存分布

改集合. fun1 在指令 0x4113C4 处将 0 写入分层偏移地址（$st1+4$）@4@0. 处理解引用是已经知道（$st1+4$）@4 上储存的是指针 $st.j$，对其解引用，获得了地址 $sf2-20$，NPtrChecker 将这个地址上储存的变量 np 的值改为 NULL，当指针 np 在 0x411426 这条指令处被解引用，NPtrChecker 检测模块报告了一个可疑的解引用.

空指针验证模块获得这个报告后，将 fun1、fun2 通过函数内联放入同一个程序控制流图中，通过最弱前置条件求解，验证了该错误的真实性.

4　设计与实现

在本部分，我们将分别详细介绍 NPtrChecker 检测工具的预处理模块、空指针检测模块和空指针验证模块具体设计以及在过程中遇到的挑战和采取的应对策略.

4.1　预处理模块

预处理模块包括反汇编模块和中间代码转换模块. 该模块接受二进制程序作为输入，返回转换后的中间代码程序和函数调用关系等重要信息.

4.1.1　反编译模块

二进制程序是一堆机器码与数据的集合，在对其进行静态检测之前，需要先对程序进行反汇编，并提取出相关的信息. 本文选择 IDA Pro 作为反汇编工具，IDA Pro 是目前市面上最为成熟与准确的反汇编工具，除了基础的反汇编功能，还提供了函数划分、库函数识别、生成函数调用关系图等功能，为后续分析和验证提供了相关信息.

4.1.2　中间代码转换

直接在汇编程序上进行分析是一件很复杂的工作. 首先，不同的汇编指令有上百种之多，且指令的语义复杂，例如 add 语句除完成加法操作以外，还会改变标志寄存器的值，push、pop 指令会改变 esp 寄存器的值. 其次，汇编程序中相比与源程序，缺少了很多高层语义，如函数信息、变量类型等. 所以本文选择将汇编指令等价转换为类型较少、语义简单的中间代码.

本文选择二进制分析平台 BitBlaze[14] 提供的 Vine IR 作为我们的中间语言，同时 BitBlaze 提供了将汇编程序转换为 Vine IR 的功能. 对于 IDA Pro 返回的汇编程序，BitBlaze 按照汇编指令的顺序，逐条将汇编指令转换为语义对等的 Vine IR 指令块，每个 Vine IR 指令块包含多条 Vine IR 指令. Vine IR 的主要语法如图 5 所示.

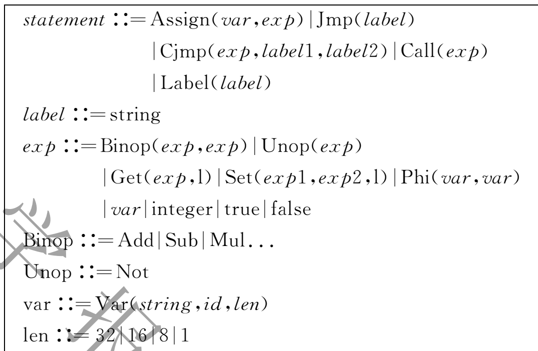

图 5　Vine IR 的语法

我们采用 SSA 形式来表示中间语言中的变量. 在 SSA 形式的中间语言中，每个被使用的变量都有唯一的定义，引用-定值链（ud 链）非常明确，可使数据流分析更加简单. Vine IR 中间语言主要包含五种不同的语句：

（1）Assign(*var*, *exp*)：将表达式 *exp* 的值赋值给变量 *var*.

（2）Jmp(*label*)：跳转到 *label* 标志的语句处.

（3）Cjmp(*exp*, *label*1, *label*2)：条件跳转语句，当条件 *exp* 为真时，跳转到语句 *label*1，否则跳转到 *label*2.

（4）Call(*exp*)：函数调用指令，调用 *exp* 表示的地址处的函数.

（5）Label(*label*)：标识位置的语句.

中间语言中有多种不同的表达式，其中 Binop(*exp*, *exp*) 和 Unop(*exp*) 分别表示二元运算

和一元运算；Get(*exp*,*len*)表示读内存操作，*exp* 是内存地址，*len* 表示内存值的位宽；Set(*exp*1,*exp*2,*len*)表示写内存操作，*exp*1 为内存地址，*exp*2 为写入的值，*len* 为内存值的位宽；除上述各类运算和内存操作以外，表达式也可以直接是变量、整数或布尔值.

二元运算 Binop 我们重点关注加法(Add)、减法(Sub)和乘法(Mul)这三个与内存地址寻址相关的运算(例如 mov *eax*, [*ecx*+4***edx*+8]). 变量 *var* 用 Var(*string*,*id*,*len*)来表示，其中 *string* 是变量名，*id* 是 SSA 模式中给每个变量的唯一标识，*len* 表示这个变量的占用了多少位宽.

4.2 空指针检测模块

在空指针检测模块，本文实现了一个过程间、上下文敏感、域敏感的数据流分析算法. 从函数入口点开始，NPtrChecker 按照指令执行顺序，沿着函数的程序控制流图收集变量的相关信息，当一个可能为空的变量被用于解引用时，我们便将这作为一个可疑的空指针解引用保存起来，并最终交给空指针验证模块进行验证.

4.2.1 内存模型

为了实现上述数据流分析，我们建立了一套内存模型来模拟指针对内存的访问操作，跟踪并记录指针的变化和指针间的关系.

我们将一个函数内能访问到的内存地址分为三类：

(1) 全局内存地址. 全局内存地址是全局变量的储存位置，通常直接通过一个立即数给出.

(2) 栈内存地址. 通过栈偏移访问的地址，一般是以"*esp*/*ebp*+*offset*"的形式访问. 栈内存地址上存储的变量包括函数的局部变量和函数的参数. 为了便于分析且不失一般性，我们设函数入口处的栈偏移为 0，根据栈生长方向我们可知，参数的栈内存地址大于 0，局部变量的内存地址小于 0.

(3) 特殊内存地址. 特殊内存地址是指危险库函数的返回值指向地址，以及被赋予 NULL 值的变量指向的地址. 我们对程序执行路径上的每一个这种地址给与一个不同的编号，用于区分它们.

我们用 location_type(s)来表示内存地址的类型，包括 GLOBAL，STACK，SPECIAL 三种，分别对应上面的三类内存地址类型.

对于需要多次解引用才能访问到的变量，假设它的上层指针的内存地址为 location_type(s)，则它对应的地址表达为：location_type(s@offset). 例如：

mov *eax*, [*esp*+4]

mov *ebx*, [*eax*+4]

由第一条语句可知，*eax* 来自栈内存地址[*esp*+4]，假设当前 *esp* 的值为 0，我们将这个栈内存地址表示为 STACK(4). 在第二条指令中，*ebx* 来自内存地址[*eax*+4]，已知上层指针 *eax* 的地址为 STACK(4)且偏移量为 4，*ebx* 的地址表示为 STACK(4@4). 我们将这种地址的表达方式称为多层偏移地址. 通过这种层级表示，可以清晰的表示相关变量间的关系.

我们建立了三张映射表来储存数据流分析中获得的信息：

MT：内存地址表. 我们定义 *MT* 为 $M_{loc} \to 2^V$ 的二元映射关系. M_{loc} 为函数中访问过的内存地址的集合；*V* 为函数中变量的集合. 对于任意 $m_loc \in M_{loc}$，$var \in V$，$var \in MT(m_loc)$ 表示 *var* 为可能储存在地址 *m_loc* 上的变量.

NT：变量空值信息表. 我们定义 *NT* 为 $V \to 2^{NR}$ 的二元映射关系. *NR* 为空值信息的集合，一条空值信息记录了变量的内存地址 *m_loc*，当前偏移量 *offset*，变量值来源的指令地址 *source_addr* 以及该变量是否为空值 is_null. 其中 *offset* 为整数类型，该变量来自内存地址 *m_loc*，指向内存地址 *m_loc*@*offset*；*addr* 为 Vine IR 中的 label 类型；is_null 为 BOOL 类型，表示该变量是否可能为空值.

DT：解引用信息表. 定义 *DT* 为 $V \to 2^{AD \times NR}$ 的二元映射关系. *AD* 为指令地址的集合.

每条指令对应一张独立的 *MT* 表和 *NT* 表，每个函数对应一张独立的 *DT* 表.

4.2.2 状态转移函数

状态转移函数描述对应语句上，程序状态对应的转移关系. 图 6 给出了我们的数据流分析中与指针操作相关几个重要语句的状态转移函数.

(1) Assign(*var*1,*exp*(*var*2))：表示将变量 *var*2 的值赋值给变量 *var*1. 对应的状态转移函数为将 *var*2 的空值信息 *NT*(*var*2)赋给 *var*1 的空值信息 *NT*(*var*1).

(2) Assign(*var*1,*exp*(Add(*var*2,*n*))：表示将变量 *var*2 加上整数 *n* 赋值给变量 *var*1. 此时需要修改 *NT*(*var*2)信息，对 *NT*(*var*2)中的任意一条空值信息 *nr*，修改 *nr*.*offset* 为 *nr*.*offset*+*n*，将修改后的值传给 *NT*(*var*1).

(3) Assign(*var*1,*exp*(Phi(*var*2,*var*3))：表示 *var*1 的值可能来自 *var*2 或 *var*3. 因此将空值信息 *NT*(*var*1)的值修改为 *NT*(*var*2)与 *NT*(*var*3)的合集.

《计算机学报》

```
1. Assign(var1,exp(var2)):
   NT(var1):=NT(var2)

2. Assign(var1,exp(Add(var2,n)):
   foreach nr in NT(var2)
     add (nr with nr.offset <- nr.offset + n) to NT(var1)

3. Assign(var1,exp(Phi(var2,var3)):
   NT(var1):=NT(var2) ∪ NT(var3)

4. Assign(var1,Get(var2,len)):
   foreach nr in NT(var2) do
     if nr.is_null==true then
       add(deref_addr,nr) to DT(nr.var)
     if len==pointer_size then
       if (nr.m_loc)@(nr.offset) is a key in MT then
           foreach var_t in MT((nr.m_loc)@(nr.offset)) do
               add NT(var_t) to NT(var1)
       else
         nvar := create_new_var
         nnr := create_new_null_information
         MT(nr.m_loc@nr.offset)={nvar}
         add nnr to NT(nvar)
         add nnr to NT(var1)

5. Assign(var1,Set(var2,exp(var3),len)):
   foreach nr in NT(var2) do
     if nr.is_null==true then
       add (deref_addr,nr) to DT(nr.var)
     if len==pointer_size then
       if (var2 point to one memory location then)
         MT(nr.m_loc@nr.offset)={var3}
       else //var2 may point to multiple memory location
         add var3 to MT(nr.m_loc@nr.offset)
```

图 6　状态转移函数

(4) Assign(*var*1,Get(*var*2,*len*))：将指针 *var*2 解引用的值赋值给 *var*1,*len* 表示变量 *var*1 占内存的大小.对于 *NT*(*var*2)中的任意一条空值信息 *nr*,我们首先检测是否可能为空值(*nr*.is_null 为真),若为空值,将当前指令所在地址 deref_addr 和空值信息 *nr* 添加到 *DT* 表中去.然后,当解引用的结果为一个指针时(*len* 等于指针的长度),更新 *NT*(*var*1)的信息:如果 *nr* 所在内存地址被访问过,我们根据 *MT* 和 *NT* 表的信息,更新 *NT*(*var*1);若 *var*1 来自函数内未访问过的地址,则新声明一个变量 *nvar* 和空值信息 *nnr*,并同步更新 *var*1、*nvar* 在 *MT* 和 *NT* 中的信息.此处 *nnr.m_loc* 的值为(*nr.m_loc*)@*nr.offset*,*nnr.offset* 为 0,*nnr.source_addr* 为当前指令地址,*nnr*.is_null 为真.

(5) Assign(*var*1,Set(*var*2,*exp*(*var*3),*len*))：往 *var*2 指向的内存空间上写入数据 *var*3,*var*3 的位宽为 *len*.首先检测 *var*2 是否可能为空指针,若是则更新 DT 表.然后当 *var*3 可能为指针变量时(*len* 等于指针的长度),判断 *var*2 是否可能指向多个内存地址,若是,则将 *var*3 添加入 *MT* 表对应的内存地址的变量集合中,否则用 *var*3 替换掉原有的变量集合.

《计算机学报》

4.2.3　内存模型上的数据流分析

算法 1 描述了建立在上节提出的内存模型上的上下文敏感、域敏感的基于 worklist 的数据流分析算法.该算法以函数 *N* 的起始指令 s_e 为输入,进行广度优先的遍历,最终返回解引用信息表 *DT* 和函数出口点处内存信息 *MT* 以及变量空值信息 *NT*.函数 *N* 可以是一个入口函数,或者动态链接库的导出函数,这两类函数的参数通常被认为是可控制的.

算法 1.　AnalyzeMethod 数据流分析框架.

输入：s_e：函数 *N* 起始指令

输出：*DT* 以及出口点的 *STATE*(*NT*,*MT*)

声明：worklist：(*s*,*STATE*)列表，FST：函数摘要表，*CS*：函数栈

```
begin
1. DT := ∅; worklist := (s_e, ∅)
2. while worklist ≠ ∅ do
3.   remove (s,STATE) from worklist
4.   if (s 不是 call 指令) then
5.     DT,STATE′=State_trans(s,STATE,DT)
6.   else if (s 调用了函数 M) then
7.     if FST(M)=∅ then
8.       push N onto CS   //分析被调函数
9.       s_x := M 的首条指令
10.      DT″,STATE″=AnalyzeMethod(s_x)
11.        DT,STATE′=ApplyFuncSummary(DT,
                 STATE,DT″,STATE″); pop CS
12.    else
13.      DT″,STATE″=FST (M)
14.        DT,STATE′=ApplyFuncSummary(DT,
                 STATE,DT″,STATE″)
15.  foreach successor s_s of s do
16.    if (s_s,STATE″) is not in worklist then
17.      add (s_s,STATE′) to worklist
18.    else
19.      MSTATE=MEET(STATE′,STATE″)
20.      replace(s_s,STATE″) with (s_s,MSTATE) in
         worklist
21. FST(N)=(DT,STATE) //函数出口点的 STATE
22. if lenth(CS)=0 then report DT
23. return (DT,STATE)
end
```

第 1 行到第 20 行描述了从进入函数 *N*,到退出函数 *N* 的所有分析过程.worklist 储存着(*s*,*STATE*)序列,其中 *s* 是待分析的语句,*STATE* 则是语句 *s* 入口处状态信息.*STATE* 包括了此时的内存信息 *MT* 和变量空值信息 *NT*.

分析程序在第 2～20 行迭代的处理 worklist 中的元素.对于取出的每个(*s*,*STATE*),我们检测语

句 s 的类型，如果不是 call 指令，则按照上一小节中的状态转移函数来处理(第 4～5 行). 如果是一个 call 指令，检测是否存在被调函数 M 的函数摘要，如果有，直接使用摘要(第 12～14 行)；如果没有，调用 AnalyzeMethod 分析函数 M，获得函数摘要(第 23 行)后再使用这个摘要(第 7～11 行).

函数 M 的函数摘要包括以下两部分：函数 M 及其调用过的函数中的所有可能的空指针解引用的信息，这些信息都保存在 DT 表中；以及函数 M 出口点处理后的 STATE 状态，STATE 中不保留储存在局部栈上的变量的信息，因为这些变量会随着函数调用的返回被释放掉.

第 11 行和第 14 行的 ApplyFuncSummary 提供了函数摘要应用的功能，算法 2 通过伪代码描述了该算法的过程. ApplyFuncSummary 实现了两个功能：处理被调函数中的空指针解引用信息(1～6 行)和被调函数对主调函数内存值的修改(7～12 行).

算法 2. ApplyFuncSummary 算法.

```
输入：主调函数中的 DT、STATE，被调函数的 DT″、STATE″
输出：经过修改后的 DT、STATE
声明：(MT,NT)=STATE,(MT″,NT″)=STATE″
begin：
1. foreach (var,di_list) in DT″ do
2.   foreach (deref_addr,nr) in di_list do
3.     caller_nrl=get_caller_nrl(nr.m_loc,nr.offset)
4.     foreach cnr in caller_nrl do
5.       if cnr.is_null==true do
6.         add (deref_addr,cnr) to DT(cnr.var)
7. foreach (mem_loc,vars) in MT″ do
8.   caller_loc_list=get_caller_loc_list(mem_loc)
9.   add vars to MT(caller_loc_list)
10.  foreach var in vars do
11.    caller_nrl=get_caller_nr_by_var(var)
12.    NT(var)=caller_nrl
end
```

在处理被调函数中的空指针解引用信息时，我们查找主调函数中对应的空值信息(第 3 行)，当被解引用指针可能为一个空指针时(第 5 行)，将信息添加入主调函数的 DT 表中(第 6 行). 核心是第 3 行的 *get_caller_nrl* 函数，如果空值来源于 SPECIAL 类型地址，直接返回该空值信息；如果来源于 STACK 或 GLOBAL 类的地址，我们可以通过多层偏移地址的方式，找到主调函数中对应的信息. 例如，假设空值来自被调函数的 $STACK(a@b)$ 内存地址，且当前 $offset$ 值为 o. 我们首先查找主调函数的 $STACK(a+stk_o)$ 内存位置处的变量对应的空值信息，其中 stk_o 为主调函数与被调函数之间的栈偏移. 设找到的空值信息为 $nr1$，则我们再查找 $STACK(nr1.m_loc@(nr1.offset+b))$. 设第二次找到的空值信息为 $nr2$，修改 $nr2.offset$ 为 $(nr.offset+o)$，这便是主调函数中对应的空值信息.

处理被调函数对主调函数内存值的影响. 我们首先找出被调函数中内存地址对应主调函数中的哪些内存地址，并更新主调函数的 MT 表(第 8，9 行). 对于这些被调函数 MT 表中的变量，我们查找主调函数中对应的空值信息，并更新主调函数的 NT 表(第 11，12 行). 第 8 行和第 11 行的函数实现方式与 *get_caller_nrl* 类似.

算法 1 的第 15～20 行描绘了数据流信息的交汇. MEET 函数将来自不同分支语句的 $STATE$ 信息合并. 对于 NT 表，直接取并集. 对于 MT 表，若不同 MT 表的同一多层偏移地址上只有其中一个有变量，表示在其中一条路径上这个内存地址上的存储的值被读取过或修改过，则新建一个变量，表示这个地址上原始的变量，然后再取交集.

我们的数据流分析算法不处理函数的循环调用和递归调用函数(第 8 行). 在对目标函数 N 完成分析后，数据流分析模块将分析结果传给验证模块(第 22 行).

除此之外，先检测一个指针是否为空，然后根据结果判断是否进行解引用，这是编程人员为预防空指针解引用而采取的常见保护手段. 我们的数据流分析对这种情况做了简单有效的处理.

4.2.4 库函数建模

内存分配、字符串处理、文件操作等库函数可能会返回 NULL 指针，或解引用传入的指针类型的参数，如果不能正确识别并处理这类函数，将会导致漏报.

本文参考库函数的接口定义，通过人工建立函数摘要的方式，对以上三类库函数进行了建模. 库函数的函数摘要描述了该函数对传入指针参数的要求、返回值的状态和对内存的修改.

当空指针传递给一个要求为非空的库函数参数，或者库函数返回的空指针被直接解引用时，检测模块会报告一个空指针解引用错误.

4.2.5 局限性

尽管本文的方法实现了基于函数摘要的上下文敏感、域敏感的数据流分析，现在仍有一些问题是暂时没有解决的. 首先，像现有的空指针检测工具一样，NPtrChecker 不能有效分析函数的递归调用和交

《计算机学报》

互递归.对于这类函数，我们只分析一遍，当函数通过递归或交互递归的方式调用自己时，NPtrChecker跳过对该次调用的分析，并将该次调用的返回值设为非空，然后继续进行检测.其次，在遇到循环体结构时，我们只进入循环一次.这些劣势可能会导致分析结果中出现误报和漏报.

4.3　空指针验证模块

由于数据流分析未考虑路径条件的可满足性，因此空指针检测模块给出的结果中会存在较多的误报，需对这些结果进行验证.

空指针验证模块的输入为每个函数的解引用信息表DT.对于每条待验证的空指针解引用信息，我们首先找出从空指针产生点到解引用点的函数调用序列；然后通过函数内联将这些函数展开并连接到同一个控制流图中，调用序列上的递归函数和交互递归函数只展开一次，避免重复无限内联；最后收集空指针解引用需要满足的最弱前置条件，并调用约束求解器判断条件是否能被满足.

对收集到的最弱前置条件进行约束是整个检测流程中最耗时的步骤，我们采用以下几种方法来减少约束求解器的调用，提高整体效率.

首先，对于非入口函数，NPtrChecker只检测SPECIAL类型内存地址上空指针解引用，即只检测危险库函数返回值或NULL赋值语句导致的解引用.这不代表DT表中的STACK或GLOBAL类型地址上的空指针解引用没有意义.例如，函数可能将一个SPECIAL内存地址类型的空指针的通过参数压栈或全局变量传递给被调函数，并在被调函数中被解引用.在被调函数的DT表中，这是一个STACK/GLOBAL上的空指针解引用；主调函数在使用了被调函数的函数摘要后，会找到对应的SPECIAL内存地址，并报告一个SPECIAL上的空指针解引用.

其次，对于任意函数，我们根据该函数的控制流图生成对应的支配树.如果同一个空指针在多个不同的位置被解引用，且其中一个解引用位于在其他解引用的必经路径上（即该解引用点支配其他点），则我们只检测位于该必经点的解引用.

最后，一条产生空指针解引用的路径可以分为两个部分：从函数入口点到空指针产生点的路径和从空指针产生点到解引用点的路径.我们首先对函数入口点到空指针产生点的路径进行验证：若约束求解器无法计算出有效解，则不再对具有相同前半段路径的可疑错误进行验证；反之，则对整条路径进行验证.

5　实验评估

我们实现了空指针检测工具NPtrChecker，并实现了本文中描述的所有算法.据我所知，现阶段除本工具以外没有其它面向二进制程序的空指针检测工具，故本节将NPtrChecker和面向源代码的空指针解引用检测工具做对比实验.

我们在一台使用1.6 GHz主频英特尔12核24线程XEON处理器，内存为64GB的服务器上进行单进程实验.实验用例参考LUKE等工具的测试集，选择对SPEC2000的CINT2000中的全部11个C程序进行检测.SPEC2000是CPU性能评估的测试工具集，包含多个测试程序，因为跨平台、代码质量高、并提供源代码，经常被各类工具选为测试用例.我们将实验结果与Saturn（使用1.2版本，并采用其说明文档中提供的默认选项）和LUKE这两个开源的空指针检测工具进行对比.据我们所知，这两个工具是目前我们能获得的最优秀的空指针检测工具.我们分析了各个工具检测精度（误报率和相对漏报率）和检测效率，并对二进制和源代码检测表现出的不同进行了分析.

5.1　检测精度分析

表1给出了不同测试工具对各个测试用例的检测结果，包括测试用例的规模（源代码行数和二进制程序的大小），报告的错误数、实错数、误报数和相对漏报数.其中报告的错误数是指检测工具报告的空指针错误的数量；我们通过人工对报告的结果进行分析，得出实错数和误报数；相对漏洞数指的是在该工具中未能被检测出、而在其它工具中被检测到的错误的数量.由于这几个检测工具都不能保证检测出程序中的所有空指针错误，所以无法给出绝对漏报数.一个空指针值可能在同一个函数的多个位置被解引用，对于这种情况，我们仅统计为一个错误.

在这11个程序中，三个检测工具共发现了22个真实错误.其中Saturn报告了92次，共发现13个真实错误，误报率为86%，相对漏报率为41%；LUKE报告3次，发现2个真实错误，误报率为33%，相对漏报率为91%；NPtrChecker报告了37次，发现了22个真实错误，误报率为41%，相对漏报率为0.在三款工具中，我们的工具发现了最多的空指针解引用错误，没有漏报其他工具中发现的真实错误，并保持了相对较低的误报率.

计算机学报

表 1 空指针检测结果

程序	源代码规模(KLOC)	二进制大小/KB	Saturn				LUKE				NPtrChecker			
			报告	实错	误报	相对漏报	报告	实错	误报	相对漏报	报告	实错	误报	相对漏报
164.gzip	8.6	86	0	0	0	0	0	0	0	0	0	0	0	0
175.vpr	17.7	238	2	1	1	0	1	1	0	0	2	1	1	0
176.gcc	288.4	1237	44	12	32	0	0	0	0	12	19	12	7	0
181.mcf	2.4	82	0	0	0	1	0	0	0	1	1	1	0	1
186.crafty	21.2	287	0	0	0	0	1	0	1	0	0	0	0	0
197.parser	11.4	166	8*	0	8	1	0	0	0	1	2	1	1	0
253.perlbmk	86.2	626	3	0	3	7	1	1	0	6	8	7	1	0
254.gap	71.4	456	10	0	10	0	0	0	0	0	3	0	3	0
255.vortex	67.2	556	23	0	23	0	0	0	0	0	2	0	2	0
256.bzip2	4.7	92	—	—	—	—	0	0	0	0	0	0	0	0
300.twolf	20.5	263	2	0	2	0	0	0	0	0	0	0	0	0
总计	599.7	4089	92	13	79	9	3	2	1	20	37	22	15	0

注：表中"*"表示 Saturn 在分析 parser 时，报告了 8 个空指针解引用错误后，程序出错并退出；"—"表示 Saturn 在完成对 bzip2 分析前，程序出错并退出.

通过对表 1 实验结果进行分析，主要有以下几方面的原因使得 NPtrChecker 结果优于 Saturn 和 LUKE：

(1) 更为精确的过程间分析. Saturn 使用函数摘要来处理过程间的分析，它将指针的状态与对应的约束条件结合起来，对于一些路径约束过于复杂的函数，Saturn 无法生成有效的函数摘要，在分析时忽略掉该次调用，导致了过多误报. 如图 7(a)所示情况，由于 Saturn 无法生成 NewBag 的函数摘要，分析时直接跳过了第 2 行的语句，所以导致了指针 hdDiff 的误报. 而 NPtrChecker 是在数据流分析时建立的域敏感、上下文敏感的函数摘要，所以我们的函数摘要没有路径约束，能对所有函数生成摘要. 路径的可达性我们放在验证模块进行约束求解，仅当有解时我们才认为它是真实的，避免了此类误报. 同时，LUKE 在检测程序时，只对同一个文件中的函数进行过程间的分析，所以导致了大量的漏报.

(2) 更为合理的检测策略. 当原函数中存在空指针判断语句时(形如：if (*ptr*==null))，Saturn 假设该指针在函数入口处初始化为空，然而事实情况并非总是如此. 如图 7(b)，Saturn 报告第 1 行函数 *bcmp* 的参数 *p* 可能为空指针，因为第 3 行指针 *p* 进行了空值检查. 而事实上第 3 行的 *p* 来自第 2 行的函数 compare_constant_1 的返回值，与第 1 行的指针 *p* 不是同一个. 而 NPtrChecker 使用了 SSA，将第 2 行重新赋值后的 *p* 视为新的指针. 此外在数据流分析时，会在我们提出的内存模型上更新新的指针与内存地址之间的映射关系. 所以我们能排除这类误报.

(3) 对库函数的处理. Saturn 未分析库函数，无法检测库函数导致的空指针解引用. 如图7(c)所示，

(a)
```
254.gap:
 1. TypHandle hdDiff=0;
    …
 2. hdDiff=NewBag(T_AGWORD,SIZE_HD+(2*len+1)*SIZE_SWORD);
 3.  *PTR(hdDiff)=hdP;
```

(b)
```
176.gcc:
 1. if (bcmp (…,p,…)) return 0;
    …
 2. p=compare_constant_1(TREE_REALPART(exp),p);
 3. if (p==0) return 0;
```

(c)
```
253.perlbmk:
 1. sMsg=(char*)LocalAlloc(0,…);
 2. dwLen=print(sMsg,"Unknown error …");
```

(d)
```
perlbmk:
 1. Newz(702,mg,1,MAGIC);
 2. mg->mg_moremagic=SvMAGIC(sv);
perlbmk 对应的汇编代码:
 .text:430FBE call malloc
 .text:430FC3 add esp,4
 .text:430FC6 test eax,eax
 .text:430FC8 jnz loc_430FF1
 .text:430FCA cmp byte_4A0B80,al
 .text:430FD0 jnz short loc_430FEF
 …
 .text:430FEF xor eax,eax
 .text:430FF1 xor ecx,ecx
 .text:430FF3 mov [eax],ecx
```

图 7 程序片段

《计算机学报》

LocalAlloc 函数可能返回 NULL，这个值会在 sprintf 函数中被解引用，产生一个空指针漏洞. Saturn 会漏报这个错误. 而 NPtrChecker 对此类函数进行了建模，因而能够检测到这类漏洞.

以上三个方面使得 NPtrChecker 获得了更好的实验结果. 此外，由于 NPtrChecker、Saturn、LUKE 都未能有效处理程序中的递归调用和交互递归等情况，所以在实验中都未能检测出这两类调用导致的空指针错误. Saturn 采用了与我们工具类似的做法，当函数通过递归或交互递归调用自己时，略去对该次调用的分析；LUKE 在论文中未描述对这两类调用的处理，同时实验结果未显示出它有处理这类调用的能力. 经过我们的统计，这些函数约占 SPEC2000 测试程序函数总数的 4.3%，可能存在一些未知错误.

5.2 检测时间开销分析

表 2 给出了 Saturn、LUKE、NPtrChecker 对 SPEC2000 中各个程序的检测时间. 其中 NPtrChecker 总共耗时 30171 s；Saturn 耗时 61252 s，为 NPtrChecker 的 2.03 倍；LUKE 耗时 25336 s，为 NPtrChecker 的 0.84 倍. 三个工具都是路径敏感的检测方法，因而检测时间并没有出现数量级的差别.

表 2 空指针检测时间开销

程序	源代码规模 (KLOC)	二进制大小/KB	检测时间/s		
			Saturn	LUKE	NPtrChecker
164.gzip	8.6	86	219	186	57
175.vpr	17.7	238	824	274	612
176.gcc	288.4	1237	27843	9187	15944
181.mcf	2.4	82	210	16	133
186.crafty	21.2	287	432	788	967
197.parser	11.4	166	650*	528	652
253.perlbmk	86.2	626	9320	8946	3689
254.gap	71.4	456	7891	1581	2637
255.vortex	67.2	556	11793	2925	4644
256.bzip2	4.7	92	35*	146	70
300.twolf	20.5	263	2035	759	766
总计	599.7	4089	61252	25336	30171

注：表中“*”表示程序未完成完整的分析.

《计算机学报》

NPtrChecker 在检测速度上没有取得明显的优势，主要有以下几点原因：首先，NPtrChecker 实现了过程间的路径敏感分析，而 Saturn、LUKE 都只是过程内敏感的，NPtrChecker 生成的路径约束条件更复杂，求解更耗时. 其次，NPtrChecker 面向二进制程序，二进制程序更为复杂(位运算、标致寄存器的操作、高层语意的缺失)，同样路径生成的约束条件更为复杂. 最后，NPtrChecker 能检测出更多的可疑空指针错误，因而需要验证更多路径的可达性.

NPtrChecker 总共发现了 22 个空指针解引用错误，平均每 1371 s 检测出一个空指针错误；Saturn 平均 4711 s 检测出一个错误，为 NPtrChecker 的 3.44 倍；LUKE 平均 12668 s 检测出一个错误，为 NPtrChecker 的 9.24 倍. 从平均发现一个错误的时间开销上来说，NPtrChecker 为三个工具中效率最高的.

5.3 面向二进制程序的检测方法的优势

对于面向源代码的检测方法，一些二进制分析中的独有优势，是现阶段面向源代码的分析所无法获得的.

编译优化是计算机领域长期研究的一个重要方向，很多技术被用于提高代码效率，减少代码体积. 相对于源代码，二进制代码通常更为精简，这是因为编译器通过常量传播、函数内联等方法，简化了程序的结构，删除了不可达的语句. 同时，源代码出于程序可读性、可维护性、可复用性等方面的考虑，通常会写得比较臃肿.

如图 7(d)所示，第 1 行 Newz 函数为结构体 mg 分配内存空间，Newz 可能失败导致 mg 指向 NULL，此时第 2 行解引用 mg 导致空指针错误. 初看可能觉得这个空指针解引用并不复杂，但是实际情况却复杂得多. Newz 函数间接调用了 perlbmk 的通用分配函数 safemalloc，该函数处理来自其他函数的各类内存分配请求，因而有大量的分支路径和保障性代码，导致与 Newz 函数相关的函数多达数十个，总代码超过了 1000 行，大大超过了现有工具的分析能力.

与此不同的，Newz 对应的汇编代码仅仅只有不到 10 行. 这是因为程序在调用 Newz 的时候，参数是确定的，因而执行路径也是确定的，编译器根据

这些信息进行了相应的优化,编译链接后得到了一段很短的汇编代码. NPtrChecker 面向二进制代码,天生能够很好的利用编译器提供的优化信息,这是我们工具检测出更多空指针解引用错误的一个重要原因.

6 结束语

空指针解引用错误是程序中常见的一类错误,现有研究大多数从源代码层检测这类问题. 相对于二进制代码,源代码提供了变量名、数据类型等高层语义信息,方便了指针分析. 但是,程序的源代码不一定能够获取,在一定程度上限制了这些方法的应用. 此外,面向二进制的分析能够更充分的利用编译器提供的优化信息,分析结果甚至能够超过源代码上的分析.

本文提出了一种面向二进制程序的空指针解引用错误检测方法,并实现了检测工具 NPtrChecker. NPtrChecker 在二进制上实现了上下文敏感、域敏感的指针分析;对于可能为空的指针,我们通过数据流分析检测该指针是否被解引用,并利用最弱前置条件验证这些错误是否能够被真实触发. NPtrChecker 输出空指针的产生地址、解引用地址、函数调用序列等相关信息,帮助安全分析人员和软件开发者完成对错误的定位以及后续分析. 实验表明,在克服了二进制上分析的困难并利用了优势之后,NPtrChecker 可以提供更优于源代码分析的结果.

下一步我们将提升工具的检测效率;并尝试将检测方法运用到未初始化内存访问,栈内存的 use-after-free 等错误的检测上去.

参 考 文 献

[1] Xie Yi-Chen, Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability. ACM Transactions on Programming Languages and Systems, 2007, 29(3): 16

[2] Aiken A, Bugrara S, Dillig I, et al. An overview of the Saturn project//Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. San Diego, USA, 2007: 43-48

[3] Xie Yi-Chen, Chou A, Engler D. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Software Engineering Notes, 2003, 28(5): 327-336

[4] Babic D, Hu A J. Calysto: Scalable and precise extended static checking//Proceedings of the 30th International Conference on Software Engineering. Leipzig, Germany, 2008: 211-220

[5] Nanda M G, Sinha S. Accurate interprocedural null-dereference analysis for Java//Proceedings of the 31st International Conference on Software Engineering. BC, Canada, 2009: 133-143

[6] Madhavan R, Komondoor R. Null dereference verification via over-approximated weakest pre-conditions analysis//Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Portland Oregon, USA, 2011: 1033-1052

[7] Wang Qian, Jin Da-Hai, Gong Yun-Zhan. Null dereference detection via a backward analysis//Proceedings of the 20th Asia-Pacific Software Engineering Conference. Ratchathewi, Thailand, 2013: 553-558

[8] Hovemeyer D, Spacco J, Pugh W. Evaluating and tuning a static analysis to find null pointer bugs//Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Lisbon, Portugal, 2005: 13-19

[9] Hovemeyer D, Pugh W. Finding more null pointer bugs, but not too many//Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. San Diego, USA, 2007: 9-14

[10] Dillig I, Dillig T, Aiken A. Static error detection using semantic inconsistency inference//Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. San Diego, USA, 2007: 435-445

[11] Engler D R, Chen D Y, Chou A. Bugs as inconsistent behavior: A general approach to inferring errors in systems code//Proceedings of the Symposium on Operating Systems Principles. Banff, Canada, 2001: 57-72

[12] Ma Sen, Jiao Ming-Yang, Zhang Shi-Kun, et al. Practical null pointer dereference detection via value-dependence analysis//Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops. Gaithersburg, USA, 2015: 70-77

[13] Balakrishnan G, Reps T W. WYSINWYX: What you see is not what you execute. ACM Transactions on Programming Languages and Systems, 2010, 32(6): 23

[14] Song D, Brumley D, Yin H, et al. BitBlaze: A new approach to computer security via binary analysis//Proceedings of the 4th International Conference on Information Systems Security. Hyderabad, India, 2008: 1-25

[15] Guo B, Bridges M J, Triantafyllis S, et al. Practical and accurate low-level pointer analysis//Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization. San Jose, USA, 2005: 291-302

《计算机学报》

FU Yu, born in 1988, Ph. D. candidate. His research interests include program analysis and system security.

DENG Yi, born in 1980, Ph. D., assistant professor. His research interests include program analysis and system security.

SUN Xiao-Shan, born in 1979, Ph. D., assistant professor. His research interests include program analysis and system security.

CHENG Liang, born in 1982, Ph. D., associate professor. His research interests include program analysis and system security.

ZHANG Yang, born in 1971, Ph. D., associate professor. Her research interests focus on system security.

FENG Deng-Guo, born in 1965, Ph. D., professor. His research interests include information security and cryptology.

Background

With the rapid development of information technology, software reliability is becoming more and more important. Null pointer dereference may cause program crash, and allow attackers to bypass security logic or reveal sensitive information in the system, therefore it has became a research hotspot in computer security. Many approaches have been proposed to mitigate this bug. Those approaches largely fall into two categories: path-sensitive approaches leverage symbolic execution or weakest precondition to collect path conditions and filter out the unreachable paths, while path-insensitive approaches use dataflow analysis or pattern match to infer null pointer bugs. The latter ones are much more efficient while the former ones are more accurate.

However, all these tools perform analysis on the source code, which will limits their usage scope when source code is not available. Compare with source code analysis, binary analysis is thought to be more difficult because binaries lack of high-level semantics such as variable name and structure type. What's more, there is a variety of different binary instructions and many of them have sophisticate semantics, making it hard to directly perform analysis on those binaries.

This paper presents an approach to detect null pointer dereferences in binaries. We design a function-summary based, inter-procedural, context- and field-sensitive dataflow analysis algorithm to find potential bugs. To eliminate false positives as many as possible, weakest precondition is employ to filter out the unreachable paths reported by dataflow analysis. We compared our tool with state-of-the-art source code based detection tools, Saturn and LUKE, and the result shows our tool finds more null pointer dereference bugs with a relatively low false positive rate. As far as we know, our tool is the first static analysis tool to detect null pointer dereference on binaries.

This research was supported in part by the National Natural Science Foundations of China (Grant No. 61471344), the National Information Securing 242 Project of China (Grant No. 2016A086).

《计算机学报》