软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(6):1600−1618 [doi: 10.13328/j.cnki.jos.005994]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

垂悬指针检测与防御方法*

王 豫[1], 高凤娟[1], 马可欣[1], 司徒凌云[1], 王林章[1], 陈碧欢[2,3,4], 刘 杨[5], 赵建华[1], 李宣东[1]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](复旦大学 计算机科学技术学院,上海 200433)

[3](上海市数据科学重点实验室(复旦大学),上海 200433)

[4](上海智能电子系统研究所(复旦大学),上海 200433)

[5](School of Computer Science and Engineering, Nanyang Technological University, Singapore)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

摘 要: 随着技术的发展,信息物理融合系统(cyber-physical system,简称 CPS)在生活中扮演着越来越重要的角色,例如电力系统、铁路系统.如果 CPS 遭到攻击,将对现实世界的正常运转造成巨大影响,甚至威胁生命安全.垂悬指针是指向的区域被释放后未被置为空的指针,它是一种会导致攻击的软件缺陷.由垂悬指针导致的 use-after- free 和 double-free 漏洞能够执行任意恶意代码.迄今为止,只有少量工作针对垂悬指针进行检测、防御.其中多数都会导致过高的额外运行时开销.提出 DangDone 用于检测和防御垂悬指针.首先,通过静态分析检测潜在垂悬指针;然后,基于检测到的垂悬指针信息和一系列预定义的指针变换规则,依据指针传播信息变换指针,使得指针及其别名都指向同一个新引入的指针.基于该方法,实现了 DangDone 的原型工具.基于 11 个开源项目和 SPEC CPU benchmark 的实验结果表明:DangDone 的静态分析部分只有 33%的误报率,指针变换部分只引入了 1%左右的额外开销.同时,DangDone 成功防护了 11 个开源项目中的 use-after-free 和 double-free 漏洞.实验结果体现了 DangDone 的高效率及有效性.

关键词: 垂悬指针;程序转换;程序漏洞;静态分析

中图法分类号: TP311

中文引用格式: 王豫,高凤娟,马可欣,司徒凌云,王林章,陈碧欢,刘杨,赵建华,李宣东.垂悬指针检测与防御方法.软件学报, 2020,31(6):1600−1618. http://www.jos.org.cn/1000-9825/5994.htm

英文引用格式: Wang Y, Gao FJ, Ma KX, Situ LY, Wang LZ, Chen BH, Liu Y, Zhao JH, Li XD. Detecting and preventing dangling pointers. Ruan Jian Xue Bao/Journal of Software, 2020,31(6):1600−1618 (in Chinese). http://www.jos.org.cn/1000-9825/5994.htm

Detecting and Preventing Dangling Pointers

WANG Yu[1], GAO Feng-Juan[1], MA Ke-Xin[1], SITU Ling-Yun[1], WANG Lin-Zhang[1], CHEN Bi-Huan[2,3,4], LIU Yang[5], ZHAO Jian-Hua[1], LI Xuan-Dong[1]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](School of Computer Science, Fudan University, Shanghai 200433, China)

[3](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200433, China)

[4](Shanghai Institute of Intelligent Electronics & Systems (Fudan University), Shanghai 200433, China)

[5](School of Computer Science and Engineering, Nanyang Technological University, Singapore)

* 基金项目: 国家重点研发计划(2017YFA0700604); 南京大学优秀博士研究生创新能力提升计划 B

Foundation item: National Key R&D Program (2017YFA0700604); Program B for Outstanding PhD Candidate of Nanjing University

本文由“信息物理系统软件设计自动化”专题特约编辑卜磊教授、陈铭松教授、朱祺教授、刘超教授推荐.

收稿时间: 2019-08-08; 修改时间: 2019-10-23; 采用时间: 2020-01-13; jos 在线出版时间: 2020-04-18

Abstract: Due to rapid technology advance, cyber-physical system (CPS) plays increasingly important rules in society, such as power system and railway system. However, if these systems are attacked, it would be a serious problem for the world even threats human lives. Dangling pointers is such kind of software defects and can lead to use-after-free and double-free vulnerabilities, which can be leveraged by attackers. So far, only a few approaches have been proposed to protect against dangling pointers, while most of them suffer from high overhead. This paper study proposes a lightweight approach, named DangDone, to detect dangling pointers dynamically. Built upon the root cause of a dangling pointer, i.e., a pointer and its aliases are not nullified but the memory area they point to is deallocated. DangDone first detects dangling pointers by static analysis and fuzzing. Based on the result, DangDone realizes the detection by inserting an intermediate pointer between the pointers (i.e., a pointer and its aliases) and the memory area they point to. Hence, nullifying the intermediate pointer will nullify the pointer and its aliases, which causes crash when encountering use-after-free or double-free. Experimental results have demonstrated that DangDone introduces negligible runtime overhead (i.e., around 1% on average) on SPEC CPU benchmark and is able to protect 11 real-world use-after-free or double-free vulnerabilities. The evaluation demonstrates the efficiency and effectiveness of DangDone.

Key words: dangling pointers; program transformation; vulnerabilities; static analysis

信息物理融合系统(cyber-physical system,简称 CPS)是计算过程和物理过程的融合系统.它强调信息世界与物理世界的融合,强调对交互环境的实时监测与控制.它通过与交互环境的实时交互来增加或扩展新的功能,以安全、可靠和实时的方式检测或者控制一个物理实体.当前,CPS 系统被广泛应用于重要基础设施的监测与控制、航天与空间系统、电力系统、铁路系统和智能家居等诸多领域[1,2].CPS 系统中,不同异构组件的组合引起系统行为极为复杂[3].作为使命攸关与安全攸关的系统,CPS 系统需要满足实时性、安全性和可用性等特性.然而,CPS 系统的基本组件多由嵌入式系统构成,该系统主要是由以 C/C++语言为代表的编程语言实现.但是,由于开发者在 C/C++语言程序中对于指针的错误使用,可能会导致 CPS 系统行为不符合预期.其中,类似于缓冲区溢出、use-after-free 和 double-free 的漏洞,会导致系统崩溃甚至执行任意恶意代码.在这 3 种缺陷中,后两个缺陷都是由垂悬指针引起的.

垂悬指针(dangling pointer)是由于指针或其别名所指向的内存区域被释放但指针本身未被置空.虽然垂悬指针未被访问(解引用或释放)时是安全的,但开发人员可能会无意中使用垂悬指针,在运行时导致潜在的可利用程序状态.攻击者可以利用 use-after-free 或 double-free 漏洞来危害程序的正确性和安全性,如程序崩溃、信息泄露、权限提升,甚至执行恶意代码[4].自 2008 年以来,此类漏洞的数量每年翻倍,并且由垂悬指针引起的漏洞越发普遍和危险[5].但是,由于复杂的函数调用和指针指向关系,开发人员很难杜绝垂悬指针.

当前,针对垂悬指针的方法有 3 种.

- 首先,基于静态分析的方法[4−6]一般有较高的误报率和可扩展性问题(即仅适用于小规模程序);
- 其次,基于动态分析的方法[7−9]十分依赖于给定的一组测试用例,并且会面临代码覆盖问题,可能有较高的漏报率.此外,这些方法通常需要大量的工作来监控程序运行时的行为,从而导致过高的额外性能开销(>100%);
- 最后,运行时防御方法[10,11]可以通过在运行时置空所有释放内存区域的指针来动态地保护垂悬指针,以免受攻击.但是,这些方法需要额外的内存和计算资源在运行时记录和分析指针指向关系,然后根据指针指向关系置空潜在垂悬指针及其别名.因此,这些方法也会产生较高的额外性能开销(>25%).

为了避免 CPS 系统的中的垂悬指针被攻击,开发者可以借助运行时防御方法,在运行时发现漏洞,从而避免该漏洞造成恶性影响.但是,当前主流[10,11]的运行时防御方法引入了较大的额外性能开销.为了解决该问题,在本文中,我们提出了名为 DangDone 的垂悬指针防御方法,通过在编译时的程序转换来定位潜在的垂悬指针,并防御 use-after-free 或 double-free 漏洞.确切地说,该方法先通过静态方法分析潜在垂悬指针,然后通过在这些检测出的指针(即潜在垂悬指针及其别名)和它们指向的内存区域之间插入中间指针来保护每个潜在的垂悬指针.因此,将中间指针置空,能使潜在垂悬指针及其别名置空,消除垂悬指针的同时避免 use-after-free 或 double-free 漏洞造成的攻击.

我们基于 LLVM[12]实现了 DangDone,该方法不需要开发人员的干预,并且能在编译时自动运行.在实验研

究中,DangDone 成功防护了 11 个开源项目中的 use-after-free 和 double-free 漏洞,体现出 DangDone 的有效性.此外,我们使用 SPEC CPU Benchmark[13]评估了 DangDone 引入的运行时开销.结果表明,运行时的开销可以忽略不计(即平均约为 1%).这使得它比目前的方法[10,11]更加有效.

总而言之,本文的创新贡献如下.

- 提出基于静态分析的垂悬指针检测方法;
- 提出了一种通过程序转换的轻量级方法来防御由垂悬指针引起的 use-after-free 和 double-free 漏洞;
- 开发了原型工具 DangDone,通过实验,体现 DangDone 的有效性和高效率.

本文第 1 节介绍垂悬指针及其相关漏洞.第 2 节介绍 DangDone 的方法.第 3 节通过实验评估 DangDone 的有效性和性能,并讨论 DangDone 的局限性,第 4 节回顾相关工作.第 5 节对本文进行总结.

1 垂悬指针及其相关漏洞

如果指针指向已被释放的内存区域(包括栈空间和堆空间内存),则它是一个垂悬指针.如果一个垂悬指针永远不被解引用,那么它是一个良性的垂悬指针;否则是一个不安全的垂悬指针[10].垂悬指针可以是堆空间垂悬指针或栈空间垂悬指针.如果指针指向一个对象并且该对象已被释放,则指针变为堆空间垂悬指针;如果指针指向栈变量并且栈变量超出其范围,指针仍然指向栈上不再有效的地址,这时指针变为栈空间垂悬指针.图 1 显示了堆空间垂悬指针的示例.在图 1(a)中,*p*1 和 *p*2 在第 8 行代码后变成垂悬指针,因为在第 8 行的内存区域被释放后 *p*1 和 *p*2 没有置空.

```
1  //Original program
2  int main() {
3    char *p1, *p2;
4    p1 = malloc(32);
5    strcpy(p1, ``hello'');
6    p2 = p1;
7    strcat(p2, ``world'');
8    free(p1);
9    //p1 and p2 become dangling pointers
10   strcat(p2, ``?'');
11   ...
12 }
```

(a) 原始程序

```
1  //Transformed program
2  int main() {
3    char *p1, *p2;
4    p1 = DDMalloc(32);
5    strcpy(*(char**)p1, ``hello'');
6    p2 = p1;
7    strcat(*(char**)p2, ``world'');
8    free(*(char**)p1);
9    *(char**)p1 = NULL;
10   strcat(*(char**)p2, ``?'');
11   ...
12 }
13 //Self-defined memory allocation
14 void* DDMalloc(unsigned size) {
15   void** p = malloc(POINTER_SIZE);
16   *p = malloc(size);
17   return (void*)p;
18 }
```

(b) 转换后的程序

Fig.1 An example of heap space dangling pointer

图 1 堆空间垂悬指针示例

虽然垂悬指针分为栈空间指针和堆空间指针,但是本文主要关注堆空间指针,因为栈空间指针很难被利用[10].但是为了程序的正确性,DangDone 在必要的时候会转换栈空间指针(在第 2.3.1 节讨论).堆空间中的垂悬指针是导致 use-after-free 和 double-free 漏洞的根本原因.释放垂悬指针时会导致 use-after-free 漏洞,并可能会导致不可预测的行为,因为内存区域可能已被其他函数重用或存储完全不同的数据.如果垂悬指针指向的内存区域被重复使用,则 use-after-free 漏洞可能会泄漏关键信息.特权升级攻击是通过与信息泄漏类似的方式实现.即:垂悬指针指向的内存区域用于检查权限,并且该垂悬指针可以对内存区域进行写入.当函数指针使用垂悬指针时,攻击者可以利用 use-after-free 漏洞来执行任意函数.常见的劫持技术是扩散恶意内容以覆盖垂悬指针所指向的堆中的函数指针或对象的虚表地址[14].这种劫持技术在 Linux 内核中容易成功,因为内核为了效率起见会更倾向于使用内存分配的重用机制,这使得攻击者更容易借助垂悬指针攻击内核[4].

use-after-free 漏洞的一个特例是 double-free 漏洞,它是由两次调用 *free* 函数引起的.double-free 漏洞可能导致内存分配函数覆盖存储在块中的内存管理信息.用 double-free 漏洞可能与利用 use-after-free 漏洞一样具有破坏性,可能导致远程代码执行[15]等问题.

2 DangDone

2.1 方法概述以及示例

2.1.1 研究框架

DangDone 建立在对垂悬指针根本原因的观察基础上:指针在它们指向的内存区域被释放后没有置空.为了将这些指针置空,我们可以在释放内存区域的语句之后插入置空操作.但是因为指针可能有许多别名,所以难以精确有效地对其进行分析.特别是静态指针指向分析不能保证准确性;动态分析需要为每个指针操作设置跟踪指令,会导致高开销[7,10].因此,在指针被释放后直接置空指针本身及其每个别名的方法不适用于实际的程序.为了解决这个问题,我们的主要想法是:在内存区域和指向内存区域的潜在垂悬指针之间插入一个中间指针,强制转换中间指针的类型,使其对潜在的垂悬指针不可见,并使所有潜在垂悬指针指向中间指针.因此,在释放内存区域的语句之后对中间指针插入置空操作,将使所有潜在垂悬指针在执行置空操作后指向 *NULL*.此时,程序在尝试解引用垂悬指针及其别名时会直接崩溃,而不会给攻击者利用 use-after-free 的机会.

基于上述观察,我们提出了一种名为 DangDone 的方法,通过编译时的程序转换自动消除垂悬指针.图 2 展示了 DangDone 的框架,它将目标程序和配置文件的源代码作为输入,并返回受保护的二进制程序.配置文件给出内存操作函数的信息(例如内存分配、释放和重新分配相关的函数名称和参数信息).随后进行缺陷检测,以静态地分析潜在垂悬指针.这些潜在垂悬指针将会作为输入传递给程序转换模块.DangDone 的核心是通过程序转换实现的,该转换在 LLVM 中间表示层[16]上进行,将目标程序变换为受保护的二进制程序.如图 2 所示,DangDone 先通过缺陷检测模块静态地分析潜在垂悬指针,再根据其结果执行指针传递和指针变换.在程序转换时,DangDone 先通过指针传递查找由其他指针传递的指针,然后基于指针变换规则在指针变换模块转换这些指针.以这种方式,DangDone 的程序转换将转换与潜在垂悬指针具有指针指向关系的所有指针.即:若别名是传递指针的子集,则该潜在垂悬指针的所有别名都将被转换.

具体来说,程序变换模块为每个潜在垂悬指针执行 4 个步骤.

(1) 将分配的指针标识为潜在垂悬指针;

(2) 构造新的分配函数,分配一个中间指针,将分配的内存地址分配给中间指针,并返回中间指针;

(3) 用新的分配函数替换原来的分配函数(如 *malloc*),转换指针解引用指令,使内存访问与原始程序一致;

(4) 最后将被释放的指针置为空.

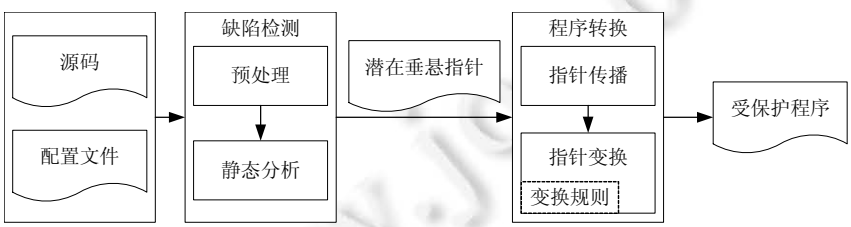

Fig.2 Framework of Dangdone

图 2 DangDone 的框架图

DangDone 不需要别名分析或指针指向分析,因为所有与潜在垂悬指针相关的指针都将被转换.通过这种方式,对潜在垂悬指针的防御被缩小到对中间指针的防御,这可以简单地通过置空中间指针来实现,因为潜在垂悬指针和它们的别名都指向那些中间指针.这些转换由 DangDone 自动完成,不需要用户干预.转换可以在源代码层、中间层或二进制层进行.虽然当前 DangDone 是在中间层实现,但是为了便于演示,我们在文中使用源代码级的示例.

2.1.2 示 例

图 1(a)显示了一个具有 use-after-free 漏洞的程序.第 10 行的漏洞源于第 8 行的内存释放.它的源码级转换

程序如图 1(b)第 2 行~第 12 行所示.这里,变量 *p*1 是潜在垂悬指针,因为它指向由 *malloc* 分配的内存区域,DangDone 将内存分配函数替换为第 14 行~第 18 行定义的新分配函数.新的分配函数包装传统的分配函数,并在分配的内存区域和用户定义的指针之间插入一个中间指针.第 6 行的指针传递语句 *p*2=*p*1 表示 *p*2 是 *p*1 的使用点(也是别名),因此应转换 *p*2.结果,两个指针都指向相同的中间指针.然后,DangDone 使用中间指针跟踪原始指针的别名(第 6 行),这样指针 *p*1 和 *p*2 上的任何读取或写入的引用都将首先访问中间指针*(char * *)*p*1,然后是*(char * *)*p*1 指向的内存区域.由于库函数 *strcpy* 和 *free* 的参数类型是 *char* *,因此,参数分别由*(char * *)*p*1 和*(char * *)*p*2 替换(第 5 行、第 8 行和第 7 行、第 10 行),以避免修改函数定义和维护原始程序的语义.DangDone 在第 9 行插入一个置空语句,以使指针*(char **)*p*1 置空,该指针的目标内存区域被显式释放.因此在第 10 行,由于 *p*2 指向*(char * *)*p*1 且*(char * *)*p*1 指向 *NULL*,*(char * *)*p*2 等于*(char * *)*p*1,程序将崩溃而不是触发 use-after-free 漏洞以及给 *helloworld* 加上?.

图 3 和图 4 展示出了原始程序和图 1 中的变换程序的别名指向关系.这里,我们在图 3 中添加置空指针的操作(即内存释放后增加一个 *p*1=*null*)以更好地说明最终的指向关系.在图 3 中,在使 *p*1 置为 *NULL* 之后,*p*2 仍然指向被释放的内存.程序员应该在释放对象时使所有别名无效,所以仅为被释放的对象提供自动置空的方法不能消除垂悬指针.相反,在图 4 中,使两个指针中的任何一个置空(即*(char * *)*p*1 或*(char * *)*p*2)都可以消除两个垂悬指针.原因是 *p*1 和 *p*2 都指向相同的中间指针 *IP*,并且置空被释放的对象将消除垂悬指针.

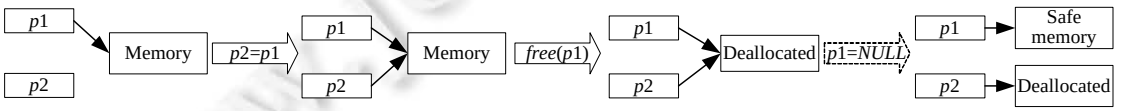

Fig.3 Point-to relations for aliases in the original program in Fig.1

图 3 图 1 中原始程序的别名指向关系

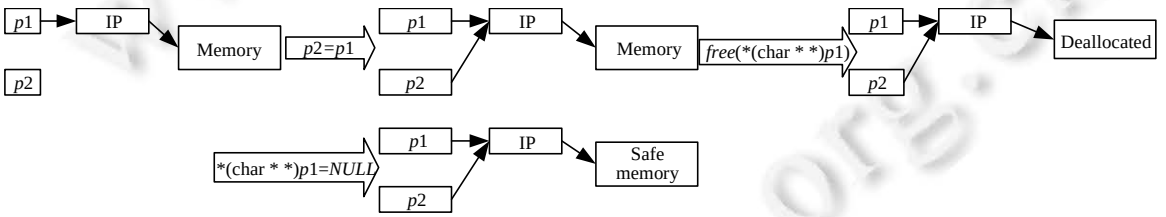

Fig.4 Point-to relation for aliases in the transformed program in Fig.1

图 4 图 1 中修改后程序的别名指向关系

为清楚起见,在下文中,我们将用原始指针指代那些易受攻击的指针,并以中间指针指代插入内存区域和原始指针之间的指针.以 *DD* 开头的函数表示这些函数是由 DangDone 定义的.变量的使用点表示直接使用变量的指令.

2.2 基于静态分析的垂悬指针检测

DangDone 进行保守的静态分析,以确定需要转换的潜在易受攻击的指针.此步骤减少了在其后的程序转换中需要考虑的指针数目.实际上,这一步是可选的,因为 DangDone 可以以效率为代价转换程序中的所有指针.

静态分析算法在算法 1 中显示,它主要作用是排除不是垂悬指针的指针.首先,它读取配置文件以获取内存释放和重新分配函数(系统的和自定义的)的列表(第 1 行),因为这些函数返回的指针可能是垂悬指针.这些内存释放和重新分配函数也会用于程序转换.然后分析程序的调用图并获得调用图的逆拓扑排序(第 2 行、第 3 行),这是为了确保在函数的调用点中,被调用者始终是被分析过的.接下来,对于每个函数,它获取此函数的所有函数调用,并标识作为内存释放或重新分配的函数调用的参数的指针(第 5 行~第 9 行).这些指针可能是垂悬指针,因为它们涉及内存释放或重新分配.

在第 2 个循环中,对于每个可疑指针,分析其别名,如果有一个指针的所有别名未都被置空,DangDone 将该指针标记为垂悬指针(第 10 行~第 16 行).在第 11 行、第 12 行,因为全局指针分析复杂,DangDone 直接将全局指针视为垂悬指针.第 16 行的 *updateMemFuns* 用于分析过程间垂悬指针.这种类型的垂悬指针是由在被调用者内,作为参数的指针被释放但没有在该函数内被置空引起的.为简单起见,DangDone 将这些函数视为一种特殊类型的内存释放函数.因此,DangDone 将在其他函数调用它们时检查对应的实参是否在调用者内被置空.第 13 行的别名分析基于 Steensgard 的算法[17].最后,在第 17 行、第 2 行,它提取潜在垂悬指针的信息并输出.所输出的潜在垂悬指针的信息包括指针名称、函数名称、代码行和文件名称.开发人员可以进一步验证这些指针,以减少误报的数量.

算法 1. 保守的静态分析.

Input: *Program*;

Output: *VulnerablePointers*.

(1) *memFuns=readConfig*(·);
(2) *functions=getCallGraph*(*Program*);
(3) *functions=getInverseTopologicalSort*(*functions*);
(4) **for** *function* in *functions* **do**
(5) **for** *callExpr* in *function.allCallExpr* **do**
(6) **if** *callExpr* not in *memFuns* **then**
(7) **continue**;
(8) *pointer=getPointer*(*callExpr*);
(9) *tmpPointers.push*(*pointer*);
(10) **for** *pointer=tmpPointers.pop*(·) **do**
(11) **if** *pointer* is *global pointer* **then**
(12) *PointerStack.push*(*pointer*);
(13) *aliasSet=analyzeMayAlias*(*pointer*,*function*);
(14) **if** not *ifAllPointersNullified*(*aliasSet*,*function*) **then**
(15) *PointerStack.push*(*pointer*);
(16) *updateMemFuns*(·);
(17) *warningList=extractPointerInfo*(*pointerStack*);
(18) *VulnerablePointers=ReportWarning*(*warningList*);

在设计静态分析时,更准确的静态分析也是可行的.例如,像 Andersens 算法[18]这样更精确的别名分析可以进一步减少误报.DangDone 也可以在没有静态分析的情况下转换所有指针,目前的保守静态分析会引入许多误报,但可以避免漏报.没有漏报的原因是垂悬指针的漏报来源于不精确的别名分析,而基于 Steensgard 的别名分析是没有漏报的.我们设计这种保守的静态分析的原因是误报只会增加开销,而漏报可以绕过 DangDone 的策略.因此在这里,我们尝试通过排除不是垂悬指针的指针来减少开销.

2.3 基于多级指针的垂悬指针防御策略

在现实世界的程序中,指针非常复杂.因此,我们研究了 SPEC CPU Benchmark[13]测试中指针的使用统计数据:27.3%的指针指向基本类型,41.7%的指针指向类或结构,16.3%的指针指向结构中的元素,6.4%的指针指向模板(包括用户定义的模板和容器),8.3%的指针是二级指针.

此外,我们根据操作是否对指针和内存产生副作用对操作进行分类:指针的副作用意味着指针指向不同的内存区域;对内存的副作用意味着内存区域被改变.

- 对指针和内存没有副作用:例如指针传递和解引用;
- 仅对指针产生副作用:例如 *malloc* 或第三方用户定义的分配函数以及指针运算;

- 副作用仅限于内存中:例如 *free* 或第三方或用户定义的释放函数;
- 对指针和内存的副作用:例如,*realloc* 或第三方或用户定义的重新分配函数.

基于上述统计和类别,我们通过图 1 和图 5 中的示例展示变换规则.其中,*ele* 表示复杂结构的类型.

```
1  //Original programs
2  //foo1: pointer propagated by function
3  ele* p, q;
4  q = malloc(12);
5  p = q;
6  foo(p);
7
8  //foo2: pointers in an array
9  //i and j are integers in [0, SIZE-1]
10 ele* a1[SIZE];
11 a1[i] = a1[j];
12 libFun(a1[i]);
13
14 //foo3: pointers in a struct
15 //sq is a global variable.
16 ele* p = malloc(sizeof(ele));
17 p->var = sq->var;
18 libFun(p->var);
```

(a) 原始程序

```
1  //Transformed programs
2  //foo1: pointer propagated by function
3  ele* p, q;
4  q = DDMalloc(12);
5  p = q;
6  foo(p);
7
8  //foo2: pointers in an array
9  //i and j are integers in [0, SIZE-1]
10 ele* a1[SIZE];
11 a1[i] = a1[j];
12 libFun(*(ele**)a1[i]);
13
14 //foo3: pointers in a struct
15 //sq is a global variable.
16 ele* p = DDMalloc(sizeof(ele));
17 (*(ele**)p)->var = (*(ele**)sq)->var;
18 libFun((*(ele**)p)->var);
```

(b) 转换后的程序

Fig.5 Examples for pointer modification rules

图 5 指针变换规则的例子

2.3.1 指针传递和解引用的规则

我们首先介绍指针传递和解引用的变换规则,它们对指针和内存没有任何副作用.

规则 1. DangDone 通过检查被分配的内存区域是否被访问来区分指针传递和解引用.如果未访问所分配的存储区,则视为指针传递;否则视为指针解引用.例如,指针传递包括指针赋值、函数参数和获取指针的地址.

规则 2. 如果指针由潜在垂悬指针传递,则指针也是潜在垂悬指针.示例如图 1 中的第 6 行,其中 *p*2=*p*1 表示 *p*2 也是潜在垂悬指针.

规则 3. 如果原始指针被解引用,DangDone 将指针转换为二级指针,并将其使用替换为额外的解引用.此时,原始指针变成了二级指针,因为新的分配函数(参见图 1 的 *DDMalloc*)与原来的分配函数不同.如图 1 中的例子所示,第 10 行中的 *p*2 由*(char **)*p*2 代替.附加的引用可以使原始程序和转换程序之间的内存保持一致.因此,如果解引用原始一级指针,DangDone 将用类型转换和原始指针的双重解引用替换原先的使用点.

对于变换后的程序,原始指针的指针传递不会改变,但是原始指针的解引用会被类型转换和附加的解引用替换.因此,中间指针对于程序的其余部分是不可见的.

然后,我们将展示如何将这些规则应用于复杂的指针,包括过程间指针、函数指针、间接指针、结构中的指针和栈空间指针.

- 过程间指针

过程间指针可以是两种类型:全局指针和作为函数参数传递的局部指针.全局指针的转换规则同局部指针.

如果原始指针作为函数参数传递,DangDone 传递原始指针而不是指向参数的间接引用指针,以使指针指向关系保持与原始程序一致.例如:在图 5 中的第 6 行有一个函数调用 *foo*(*p*),*p* 是潜在垂悬指针;并且转换后的函数调用仍然是 *foo*(*p*).但是,*foo* 的参数 *p* 与原始语义不同.因此,如果函数的参数已经转换,则需要转换相应的函数参数.在这种情况下,还应转换 *foo* 的参数以保持内存访问的一致性.然后,DangDone 搜索定位函数的所有使用点,以使它们的调用者的相应参数也被转换.这里有一个例外是处理库函数的参数时,DangDone 将参数替换为对原始指针的强制转换和解引用(如图 1 中的第 10 行).原因是当函数的参数改变时,这些参数的使用点需要进行相应的修改.因此,DangDone 不能转换没有源代码的库函数.

- 函数指针

C/C++中的一个特殊结构是指向函数的指针,这些函数指针为攻击者提供了一种通过利用 use-after-free 漏洞来注入代码的方法[4,19].据我们所知:尽管函数指针可以允许攻击者执行任意代码,但函数指针不会直接导致 use-after-free 或 double-free 漏洞,但其他类型的指针(如指向结构的指针)确实会引发此类漏洞.因此,DangDone

不处理函数指针.但是,转换的指针可以用作函数指针的参数.在这种情况下,定义满足函数指针类型的函数将进行相应的转换.

- 间接指针

我们将具有两个或以上级别的指针称为间接指针.如果我们对所有一级指针(即直接指针)进行保护,那么间接指针不会变成垂悬指针.但是,尽管间接指针不需要关心内存区域,但 DangDone 应该对它们进行转换,因为它们可以通过两个或多个解引用来访问中间指针.因此,DangDone 会分析间接指针并在它们访问中间指针时对它们进行转换,使得访问期望的内存空间.

- 结构体中的指针

对于结构体中的指针(例如数组、结构、类和模板),它们的转换遵循与其他指针相同的规则.不同之处在于解引用操作,因为访问结构的方式是多种多样的.为了使内存访问与原始程序保持一致,我们应该考虑内存访问的特性.数组中指针变换的一个例子如图 5 中的第 8 行~第 13 行所示.当 *libFun* 调用它时,DangDone 应转换指针 $a1[i]$,因为它是一个库调用.另一个显示结构中指针转换的例子如图 5 中的第 14 行~第 18 行所示.指针 $p \rightarrow var$ 被转换为二级指针并解引用,以便 *libFun* 的参数与原始程序保持一致.

- 栈空间指针及常量地址

虽然我们专注于堆空间垂悬指针,但由于指针传递,栈空间指针也可能会被转换.例如:指针在一个分支中,它指向的是分配的内存;在另一个分支中,它指向的是一个栈变量.DangDone 通过为潜在垂悬指针传递的指针构造一个中间栈指针来转换栈空间指针.中间指针的生命周期与它指向的内存区域相同.因此,栈和堆空间指针的操作是一致的.在 CPS 系统中,常量地址相较于常规应用程序更为常见.它是指针传递过程中其中一个别名被赋值成一个常量地址.针对常量地址的赋值,我们也应用相同的处理方式.示例代码如图 6 所示.

```
1  //Original program
2  void stackPointer() {
3    ...
4    int intVal = 1;
5    int* q = &intVal;
6
7    p=q;
8  }
9  void constantPointer() {
10   ...
11   int* q = 0x12345678;
12
13   p=q;
14 }
```

(a) 原始程序

```
1  //Transformed program
2  void stackPointer() {
3    ...
4    int intVal = 1;
5    int* q_1 = &intVal;
6    int* q = (int*)&q_1;
7    p = q;
8  }
9  void constantPointer(){
10   ...
11   int* q_1 = 0x12345678;
12   int *q = (int*)&q_1;
13   p = q;
14 }
```

(b) 转换后的程序

Fig.6 Examples of stack space pointer and pointer from constant address

图 6 栈空间指针和常量地址例子

2.3.2 内存分配函数的规则

现在我们介绍对指针和内存有副作用的操作规则,以便在转换后指针和内存仍然保持一致.潜在垂悬指针被释放或重新分配时,如果它们指向释放的内存,我们应该将它们置空.此外,还应考虑一个指针的多重分配.指针运算将在第 3.4 节讨论.

规则 4(内存释放). 为了防护潜在垂悬指针,如果释放了一个内存区域,DangDone 会中间指针插入置空语句.如果此时程序执行到内存释放语句和置空语句后,原始指针及其别名都最终指向 *NULL*.图 1 中的第 9 行显示了一个示例.由于所有别名都指向同一个中间指针,因此无论它们是显式释放还是隐式释放,所有别名都将置空.从这个意义上讲,DangDone 只需要为一次内存释放插入一个置空语句.

规则 5(内存重新分配). DangDone 对重新分配进行包装,以便在重新分配函数返回的指针与传递给函数[11]的原始指针不同时,置空原始指针并为新返回的指针添加一个中间指针.这个规则是由于类似于 *realloc*(·)的内存分配函数可以增加或减少所分配的内存空间.但是如果所需的内存空间太大,则重新分配的地址可能与原始地址不同.所以 DangDone 在所分配地址和原始地址不同时,置空原始指针并为新指针加入中间指针.

规则 6(内存分配和多重分配). 对于每个潜在垂悬指针,我们用 DangDone 定义的新内存分配函数替换原来的内存分配函数(包括堆和栈空间内存).新的内存分配函数在调用原内存分配函数的同时,还分配了指向内存区域的中间指针.然后返回中间指针的地址,而不是所分配的内存区域的地址.请注意:此时原始指针变为二级指针,DangDone 将类型转换为一级指针.使用新分配替换分配的目的是强制所有指针及其别名都指向中间指针.因此,该替换应该分析指针的级别(如是否是间接指针),除了在内存区域和一级指针之间插入中间指针外,应避免在其他指针之间插入中间指针.例如,图 1 中的 *DDMalloc* 展示了这种新分配的过程.关于分析指针级别的例子,如例子程序 char * * *p*=*malloc*(*size_t*);char * *p*=*malloc*(32),此转换应只替换第 2 个内存分配(即 *malloc*(32)).

多重分配是指一个程序分配了一个新的内存区域并将其分配给一个已经指向内存区域的指针.处理方式依旧遵循规则 6 以避免语义上的差异.换句话说,DangDone 不去区分某处的内存分配是否是多重分配.但是,替换分配的副作用是内存泄漏.分配引入的潜在内存泄漏问题将在第 3.4 节中讨论.

2.4 指针传递和指针变换

2.4.1 指针传递

DangDone 跟踪从原始指针传递的每个指针的指针传递,并递归地变换这些指针.它通过记录原始指针和转换原始指针的使用点来实现.原始指针的指针指向结果存储在它们相应的中间指针中,因此,指针保持着与原始程序相同的指针指向关系.

算法 2 介绍了递归变换指针的过程.输入程序是需要保护的代码.DangDone 首先分析程序,找出指向已被分配的内存区域(第 1 行)的潜在垂悬指针.然后将潜在垂悬指针存储在栈 *PointerStack*(第 2 行)中.对于栈顶部的每个指针,首先替换指针 *oriPointer*(第 5 行)的分配函数(若有)(第 4 行).相反,如果它指向栈空间(第 7 行),则为指针 *oriPointer*(第 6 行)插入一个中间指针,然后,DangDone 将原始指针标记为已修改以供进一步使用(第 8 行).然后,DangDone 基于 Def-Use Chains 和 Use-Def Chains[16](第 10 行)识别所有使用点(例如使用原始指针的指令).对于每个使用点,DangDone 通过函数 *getOtherHandSide* 获得使用点(第 11 行)也使用的指针 *otherPointer*,该函数确定与原始指针相关的指针(例如别名和解引用).由于某些指令只有一个操作数,因此 *otherPointer* 可以为 *NULL*.如果存在 *otherPointer*,则检查 *otherPointer* 是否已经转换(第 13 行);如果没有,则将此指针存到 *PointerStack*.请注意:如果未转换 *otherPointer*,则不会转换此使用点;但当两个指针都被标记为已修改时,它将被转换.在添加潜在的潜在垂悬指针后,DangDone 使用第 18 行的函数 *transformPointer* 来转换指令.如果指令是指针传递,它将保持指向关系;如果指令中未引用指针,则它将替换为其他引用.详细的转换算法见第 2.4.2 节.由于 DangDone 是在潜在垂悬指针使用到这些指针时变换这些指针,因此它不需要进行额外的别名分析.虽然递归变换可能会导致指针别名的大量转换代码带来的额外开销,但实验表明这个开销是很小的.

算法 2. 递归变换指针.

Input: *Program*;

Output: *Secured Program*.

```
(1)  Pointers=analyzeAllocatedMemory(·);
(2)  PointerStack=pushOntoStack(Pointers);
(3)  for oriPointer in PointerStack.top(·) do
(4)    if ifHasAllocation(oriPointer) then
(5)      replaceAllocation(oriPointer);
(6)    if pointsToStack(oriPointer) then
(7)      insertPointer(oriPointer);
(8)    markPointersAsModified(oriPointer);
(9)    //users(·) returns all users for a pointer.
(10)   for user in oriPointer.users(·) do
(11)     otherPointer=getOtherHandSide(user);
```

```
(12)    if otherPointer is not null then
(13)      if not isPointerModified(otherPointer) then
(14)        //This user will be modified in otherPointer.
(15)        PointerStack.push(otherPointer);
(16)        continue;
(17)    // Transform the pointer to keep memory access consistent.
(18)    TransformPointer(user,oriPointer);
(19)  PointerStack.pop(·);
```

例:对于图 1 中的程序,*PointerStack* 只包含一个指针 *p*1,因为只有这个指针被显式分配.DangDone 首先替换内存分配函数,以使原始指针 *p*1 指向中间指针.然后搜索 *p*1 的所有使用点,发现 *p*1 有 4 个使用点.因为其中 3 个没有对应的操作数,所以将他们传递到 *transformPointer*.与之对应,由于 *p*2=*p*1 有对应的操作数 *p*2,并且发现该指针尚未被修改,即 *p*2 未标记为已修改,因此,DangDone 将它存到 *PointerStack*,以便之后的指针变换.注意:即使 *p*2 不在初始的 *PointerStack* 中,DangDone 也可以保护这个指针,因为当 *p*1 的转换完成时,*PointerStack* 上的顶部变量是 *p*2,此时指令 *p*2=*p*1 将被转换.

2.4.2 指针变换

为清楚起见,算法 3 仅显示常见场景的转换.输入包括指令、指令中使用的原始指针;算法的输出是转换后的指令.首先对指令进行分析,确定操作类型.如果指令用于指针传递,则不需要转换指令.这是因为转换的一个目标是使内存访问与原始程序保持一致,并且一级指针的指针传递与二级指针相同.换句话说,对于指针传递,无论指针的级别如何,指向关系都是相同的.如果指令解引用原始指针或调用库函数,它将用另外的解引用替换原始指针的解引用(第 1 行~第 3 行).由于解引用指针的目的是访问指针所指向的内存区域,因此它添加了原始指针的额外的解引用,以使原始指针的内容与原始指令想要访问的内容一致.大多数情况下,如果原始指针解引用一次,则只需要对原始指针进行两次解引用.对于对指针或内存有副作用的操作,转换策略显示在第 4 行~第 8 行.如果指令释放原始指针,则它插入指令将中间指针置空(第 6 行).如果指令重新分配原始指针,它将使用 DangDone 定义的新的重分配替换.它添加了置空语句,根据重新分配的指针是否等于原始指针(第 8 行),将指针置为空并插入一个中间指针.

算法 3. 变换指针.

Input: *user*, *originalPointer*;

Output: transformed instructions.

```
(1)  if isPointerDereference(user) or isLibFunction(user) then
(2)    //Pointer transformation.
(3)    addAdditionalDereference(originalPointer);
(4)  else if isMemoryOperation(user) then
(5)    if isDeallocation(user) then
(6)      instrumentNullifyIns(user);
(7)    else if isReallocation(user) then
(8)      replaceReallocation(user);
```

例:继续图 1 中的程序.DangDone 分析到在转换 *p*1 时,*p*1 的 3 个使用点被传递到这个阶段.因为它使用*(char **)*p*1 作为中间指针,*p*1=*malloc*(32)转换为 *p*1=*DDMalloc*(32);*strcpy*(*p*1,“hello”)转换为 *strcpy*(*(char * *)*p*1,“hello”);*free*(*p*1)被 *free*(*(char **)*p*1)替换,并插入额外的置空操作.在转化 *p*1 之后,*p*2 的 3 种用途被传递到该阶段.两个函数调用的转换类似.*p*2=*p*1 不需要变换,因为它属于指针传递操作,并且两个指针被标记为已修改.

3 实现与实验

该方法适用于源代码、中间表示、甚至二进制.目前,为了提高效率和跨语言支持的能力,我们在基于LLVM[12]的中间表示(IR)级别实现了 DangDone(https://bitbucket.org/fff000147369/nodang).DangDone 中的转换被实现为 LLVM Pass.LLVM Pass 执行构成了编译器的转换和优化[20].因此,DangDone 的核心流程是:通过 Clang 的 AST 分析潜在垂悬指针;开发者选择潜在垂悬指针用于被防护;Clang[21]将目标项目编译为中间表示(LLVM IR[16]);DangDone 应用 LLVM Pass 来转换 LLVM IR;Clang 将转换后的 LLVM IR 编译为二进制.

3.1 实验设置

我们对 DangDone 的实验旨在回答以下 3 个研究问题.

(1) 防护能力:DangDone 是否有效地防护 use-after-free 和 double-free 漏洞?

(2) 运行时开销:就时间和内存而言,DangDone 是否有可接受的运行时开销?

(3) 垂悬指针检测效果:DangDone 的静态分析是否有足够的效果?

我们使用了 11 个真实的漏洞来评估 DangDone 的安全性,并使用 SPEC CPU Benchmark[13]来评估 DangDone 的静态分析效果和运行时开销.SPEC CPU Benchmark 是一系列 CPU 密集型程序的集合,着重于 CPU、内存以及编译器的评估.实验是在运行 64 位 Ubuntu 14.04 的 8 核 Intel Xeon CPU E5620(2.40GHz)和 16 GB RAM 的 PC 上进行的.使用 LLVM 3.6.0 以无优化(-O0)的方式编译 SPEC CPU Benchmark.并且为了展示独立的实验效果,静态分析和代码转换模块是各自单独评估的.即,此次实验的程序变换模块将所有指针视为潜在的垂悬指针.

3.2 安全性评估(RQ1)

为了评估 DangDone 对实际程序存在的漏洞的检测预防措施,我们选择在开源程序中并且有公开恶意输入(如 Exploit Database[22])的 CVE 漏洞作为基准程序.选择了 11 个 CVE 漏洞.这些漏洞的详细信息列在表 1 的前 6 列中,包括 CVE ID、受影响的程序和版本、漏洞类型以及使用的程序版本和缺陷位置.

Table 1 Evaluation results for real-world vulnerabilities

表 1 基于真实漏洞的评估结果

CVE ID	Affected program	Affected version	Vulnerability type	Tested version	Vulnerability location	Result without DANGDONE	Result with DANGDONE
CVE-2003-0015	CVS	before 1.11.5	Double free	1.11.1	server.c: 981	Application crash	Exited normally
CVE-2004-0416	CVS	1.12.x-1.12.8 and 1.11.x-1.11.16	Double free	1.12.7	server.c: 5164	Port created	Exited normally
CVE-2007-1521	PHP	before 4.4.7 and 5.x before 5.2.2	Double free	5.2.1	session.c: 1579	Segmentation fault	Exited normally
CVE-2007-1522	PHP	5.2.0 and 5.2.1	Double free	5.2.1	session.c: 850	Segmentation fault	Exited normally
CVE-2007-1711	PHP	4.4.5 and 4.4.6	Double free	4.4.5	session.c: 535	*free*(·): Invalid next size	Segmentation fault
CVE-2016-3141	PHP	before 5.5.33 and 5.6.x before 5.6.19	Use-after-free	5.5.2	wddx.c: 1030	Segmentation fault	Exited normally
CVE-2015-2787	PHP	before 5.4.39, 5.5.x before 5.5.23, and 5.6.x before 5.6.7	Use-after-free	5.6.1	zend_ execute.c: 801	Segmentation fault	Segmentation fault
CVE-2015-0273	PHP	before 5.4.38,5.5.x before 5.5.22, and 5.6.x before 5.6.6	Use-after-free	5.6.1	zend_ execute.c: 801	Segmentation fault	Segmentation fault
CVE-2012-1663	GnuTLS	before 3.0.14	Double free	3.0.13	gnutls_ pcert.c:91	Application crash	Exited normally
CVE-2009-1415	GnuTLS	before 2.6.6	Double free	2.6.5	pk-libgcrypt.c: 511	*free*(·): Invalid pointer	Exited normally
CVE-2010-2939	OpenSSL	1.0.0a, 0.9.8, 0.9.7.	Double free	1.0.0a	s3_clnt.c: 1510	Double free or corruption	Exited with code 01

与此同时,为了评估 DangDone 在 CPS 系统中的有效性,我们选取了 5 个开源 CPS 系统中注入特定的缺陷,并借助 DangDone 进行防护.所选取系统的相关信息见表 2 的前 3 列所示.所注入的缺陷信息如第 4 列~第 6 列所示.针对 Use-after-free,注入缺陷的方式是在第 6 列所指定的位置释放指针,在下个使用点之前加入指针是否为空判断语句.针对 Double free,注入缺陷的方式是在指定位置插入 *free* 语句,并在插入的语句之前加入指针是否为空判断语句.

Table 2 Evaluation results for real-world cyber-physical systems

表 2 基于 CPS 系统的评估结果

Projects	Project version	Project description	Injected bug type	Bug pattern	Vulnerability location	Result without DANGDONE	Result with DANGDONE
ardupilot	ArduPlane-4.0.0	Unmanned vehicle autopilot software suite	Use-after-free	Struct pointer	system.cpp: 531	Crash	Exited normally
PX4	v1.9.0	Flight control solution for drones	Double free	Struct pointer	bl_update.c: 152	Crash	Exited normally
FREEDM	2.0.1	A distributed grid intelligence manages power transactions and physical devices.	Use-after-free	int pointer	psocket.c: 262	Crash	Exited normally
IoT-grid	Master branch	Internet-of-Things power and energy grid	Use-after-free	Char pointer	iotgrid.c: 359	Crash	Exited normally
GridDyn	Master branch	An open-source power transmission simulation software package	Double free	two-level pointer	stackInfo.cpp: 156	Crash	Exited normally

实验结果在表 1 和表 2 的最后两列中报告,包括执行恶意输入时程序的行为以及由 DangDone 转换的程序在执行恶意输入后的行为.在表 1 中,我们发现其中 3 个程序,在其代码被 DangDone 保护后导致了段错误.原因是对空指针的解引用导致的.其余情况出现的原因则是因为在加入指针置空语句之后,原来的垂悬指针指向了 *NULL*.而这些程序会判断所使用的指针是否为 *NULL*,如果发现为 *NULL* 则会进行相应处理,如直接退出或者报错.在 DangDone 转换易受攻击的程序之后,所有的漏洞都可以在执行恶意输入之后直接退出而不会被攻击.在表 2 中,由于注入的缺陷的特征,直接执行有缺陷的结果都是程序崩溃,相比之下,借助 DangDone 防护的程序则是能够正常退出.因此,我们的实验回答了 RQ1,即 DangDone 在防护垂悬指针引起的 use-after-free 和 double-free 漏洞方面是有效的.并且,在实验过程中,我们没有发现 DangDone 在指针变换时的误报.

3.3 运行时开销(RQ2)

当 DangDone 向目标程序插入新的指令时,它会增加目标程序的执行时间和内存消耗.理论上,如果程序的所有变量都是指针,并且除了指针解引用和指针传递之外没有其他语句,那么最大运行时开销接近 100%.为了评估 DangDone 对实际应用程序的运行时开销,我们使用 SPEC CPU Benchmark 分别运行 3 次原始程序和转换后的程序来测量时间开销和内存开销.注意:由于编译问题,没有使用 PerlBench 和 Deall 这两个基准程序.

图 7 显示了 DangDone 对特定 CPU Benchmark 施加的时间开销,这些 Benchmark 的输入是在 Benchmark 中名为“Ref”目录中.我们使用几何平均数计算平均开销.平均而言,Dangdone 的时间开销为 0.3%.这表明 DangDone 引入了可忽略的运行时开销.

图 7 还显示了 DangDone 引入的内存开销.我们通过当前内存使用率的频繁快照收集内存使用率,并比较它们的最大内存使用率.所有 Benchmark 的内存开销都小于 0.8%,这表明 DangDone 引入的内存开销可以忽略不计.这是因为我们创建了很小的指针来跟踪指针.内存开销由中间指针引入,因此,DangDone 引入的内存开销取决于内存分配的数量.在大多数程序中,分配内存的大小远远大于指针的大小,导致内存开销较低.

此外,我们在表 3 中报告了规范 CPU Benchmark 的转换、指针覆盖率和编译统计信息.*#m_ptr* 表示变换的指针数,*#m_ins* 表示变换的指令数.由于指针可能有许多用途,因此变换过的指令的数量是一个更好的运行时开销指标.结合表 3 中的变换的指针、指令数和图 7 中的时间开销,我们可以看到,增加的时间开销和边换的指针、指令数都是正相关的.此外,*Dyn.Cov* 报告那些执行代码中由 DangDone 创建的所有指针中覆盖的堆指针的比例.注意,我们使用动态覆盖率是因为很难静态地区分栈指针和堆指针.平均而言,DangDone 覆盖了 83%的堆指针,

同时仍然保持着较低的时间开销.编译时,大多数 Benchmark 的开销可以忽略不计;平均而言,DangDone 在编译时预防漏洞的时间开销为 0.9%,这表明 DangDone 在编译时引入的额外开销也低.

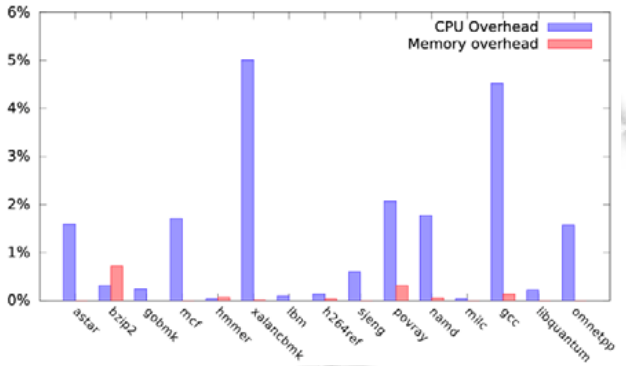

Fig.7 Overhead for the SPEC CPU Benchmarks

图 7 SPEC CPU Benchmarks 的运行时开销

Table 3 Pointer coverage and compilation statistics for the SPEC CPU Benchmarks

表 3 基于 the SPEC CPU Benchmarks 的指针转换覆盖率和编译数据

Programs	Transformation		*Dyn.Cov*.(%)	Compilation		
	#m_ptr	*#m_ins*		Orig. (s)	DANGDONE (s)	Overhead
astar	38	246	79.7	3	3	0.00%
bzip2	7	190	71.4	<1	<1	N/A
gobmk	58	736	62.1	11	11	0.00%
mcf	9	602	66.7	1	1	0.00%
hmmer	322	12 395	86.5	5	5	0.00%
xalancbmk	6 119	37 142	83.9	211	226	7.11%
lbm	4	39	100	<1	<1	N/A
h264ref	182	12 998	95.9	4	4	0.00%
sjeng	22	990	100	2	1	–50.00%
povray	905	35 732	57.8	20	21	5.00%
namd	1 298	14 117	99.9	2	3	50.00%
milc	118	2 932	80.2	2	3	50.00%
gcc	8 290	199 676	52.7	31	42	35.48%
libquantum	24	1 007	100	1	1	0%
omnetpp	538	3 542	99.9	32	41	28.13%

总之,运行时开销表明了 DangDone 的正确性,因为 SPEC CPU Benchmark 有结果比较机制,以确保结果与预先定义的结果相同.所以这部分实验表明,DangDone 的方法在 SPEC CPU Benchmark 不会对程序执行有影响.同时,图 7 和表 3 中的结果肯定地回答了 RQ2,即:DangDone 对程序的开销可以忽略不计,编译的时间开销也可以接受.

3.4 垂悬指针检测效果评估(RQ3)

表 4 展示了静态分析的效果.该分析方法分为两部分:读取 AST 的时间和分析时间.为了提升效率,DangDone 通过动态读取 AST 以优化内存开销.在该实验中,我们依旧使用 SPEC CPU 2006 Benchmarks,配置的动态读取 AST 的数目分别为 1 000 和 10 个.*#ASTs* 和*#KLOC* 是 AST 的数目和程序行数.针对这两个 AST 读取数目的配置,我们分别统计他们的读取时间,分析时间和内存消耗.*#Warnings* 是静态分析报告中潜在垂悬指针的数目.在此次实验中,我们只考虑 *free* 和 *delete* 这两个内存释放函数.*FP Rate* 是 DangDone 在特定程序下的误报率.误报是通过人工检查每个报告统计的.

我们可以看到:如果内存足够,DangDone 的静态分析是轻量级的,并且它的平均分析速度是 7*KLOC*/s 左右.尽管该静态分析工具可能消耗 4GB 的内存,但所有的程序都可以在 22s 内进行分析.相比而言,当加载在内存中的 *AST* 数被设为 10 时,分析速度会降为 1.8*KLOC*/s,但是内存消耗会降到 380Mb,几乎所有现代 PC 都可以提供

该要求.此外,不同的程序消耗不同的内存.一个原因在于 *AST* 的大小是不同的,例如有的 *AST* 大小为 30MB 而有的为 0.1MB.如果不限制内存中加载的 *AST* 的数量,则遇到超大规模的项目时,很容易消耗所有内存.

Table 4 Performance of static analysis for the SPEC CPU 2006 Benchmark

表 4 静态分析方法在 SPEC CPU 2006 Benchmark 上的性能

Programs	#*ASTs*	*KLOC*	1 000 *ASTs*			10 *ASTs*			#*Warnings*	*FP rate*
			Loading time (s)	Analysis time (s)	Memory usage (Mb)	Loading time (s)	Analysis time (s)	Memory usage (Mb)		
astar	10	4.3	<1	<1	66.1	<1	<1	62.9	7	71.40%
bzip2	7	5.7	<1	<1	61	<1	<1	61	0	0.00%
gobmk	67	157.7	5	<1	340	80	8	210	47	65.90%
mcf	11	1.6	<1	<1	100	<1	<1	80	3	0.00%
hmmer	57	22.7	2	<1	235	10	4	90	308	33.30%
xalancbmk	738	266.8	94	3	4 012	1 066	183	372	295	40.90%
lbm	2	0.9	<1	<1	53	<1	<1	53	1	100.00%
h264ref	42	36	2	<1	140	9	4	40	182	12.50%
sjeng	19	10.6	<1	<1	80	<1	<1	70	20	33.30%
povray	100	78.7	10	<1	580	86	27	133	90	33.30%
namd	11	3.9	<1	<1	87.9	1	<1	85.5	405	11.10%
milc	68	9.6	1	<1	133	3	1	80	36	0.00%
gcc	155	386.1	18	4	1 080	297	73	163	44 728	22.70%
libquantum	16	2.5	<1	<1	24	<1	<1	22	4	0.00%
omnetpp	85	26.6	6	<1	701	52	12	144	209	40.90%

此外,为了评估静态分析的误报率,如果 *warning*(告警)数小于 50,则人工验证所有 *warning*;否则,随机抽取 50 个进行人工验证.我们发现,静态分析的误报率约为 32.657%.此外,为了评估静态分析的漏报率,我们使用 Fortify SCA[23],Splint[24]以及 Coverity[25]作为基准,共报告了 106 处 *warning*.具体来说,Fortify 报告了 23 处 *warning*,包括 12 个 double-free 和 11 个 use-after-free.Splint 和 Coverity 分别报告了 72 个和 11 个 *warning*.我们人工验证这些 *warning* 并发现 DangDone 的静态分析部分也会报告所有 *warning*,这说明 DangDone 的静态分析的漏报率为 0.

综上所述,表 4 的结果可以回答 RQ3,静态分析可以通过减少潜在垂悬指针数量来减少 DangDone 的时间开销;这需要一定的时间开销以及内存开销;并且具有可接受的误报率和非常低的漏报率.

3.5 讨 论

- 漏报

据我们所知:当应用到现实项目中时,DangDone 会引入漏报,即无法防御指针计算生成的指针.因为 DangDone 试图使所有别名指向中间指针,而指针算法不遵循此规则.例如,图 8 中的转换程序创建一个生命周期与 *p*2 相同的指针,然后将指针运算的结果赋给新创建的指针.最后,它将地址存储到 *p*2,使所有转换的指针具有相同的级别.但是,第 10 行的引用仍然是 use-after-free.尽管 DangDone 无法保护通过指针运算创建的潜在垂悬指针,但比起不转换,不如对其进行转换,因为我们仍然可以保护部分漏洞(例如指针的指针算法位于分支中时).在我们的实验中,尚未发现与指针运算相关的垂悬指针.

- 间接赋值

别名可以通过间接赋值引入.对于存储在模板中的原始指针(即用户定义的模板和容器),我们递归地转换模板中使用的所有指针(即模板的使用点).如果不转换所有指针,会导致程序语义发生变化.

- 类型转换

非指针类型变量之间的指针传递可能导致特殊别名,但是 DangDone 可以识别这样的指针传递;这些类型的指向关系可以被它们对应的原始指针跟踪,它们的指针类型可以由 DangDone 确定.

- 释放中间指针

虽然 DangDone 通过替换内存分配函数分配了中间指针,但是不能通过替换内存释放函数释放中间指针.这可能导致内存泄漏.为了解决这个问题,我们通过静态分析得到的保守的指针分析结果,在某个潜在垂悬指针

及其别名中分析出生命周期最长的变量,确定其最后一次使用点所在的函数.在该函数的返回之前,插入中间指针释放语句.

- CPS 系统中的防护结果

虽然 DangDone 针对 use-after-free 和 double free 漏洞的防护结果是导致程序崩溃,而 CPS 系统中的可用性是一个重要指标,直接导致程序崩溃会影响其可用性,但是值得注意的是:在借助于来自 CVE 的漏洞的实验中,有一些程序能安全退出是通过检测被使用的指针是否为 *NULL*.所以在 CPS 系统中,虽然我们能够避免更严重的攻击后果(如权限提升、执行恶意程序等),但是依旧需要开发者有一定的容错机制.例如,在指针的使用点需要检测该指针是否为 *NULL*:如果是,则进入错误处理.

```
1  //Original program
2  char *p1, *p2;
3
4  p1 = malloc(16);
5
6  p2 = p1+3;
7  ...
8  free(p1);
9
10 *p2 = '2';
11  ...
```

(a) 原始程序

```
1  //Transformed program
2  char *p1, *p2;
3  char *m1;
4  p1 = DDMalloc(16);
5  m1 = *(char**)p1 + 3;
6  p2 = (char*)(&m1);
7  ...
8  free(p1);
9  *(char**)p1 = NULL;
10 *p2 = '2';
11  ...
```

(b) 转换后的程序

Fig.8 Dangling pointer caused by pointer arithmetic

图 8 指针计算导致的垂悬指针

4 相关工作

4.1 改进内存分配函数

我们改进了内存分配函数来保护垂悬指针.Cling[26]替换了原来的内存分配函数,因为一些攻击手段依赖于释放内存的重复使用,它会避免重复使用同一类型的内存对象.DangDone 也用自定义的内存分配函数替换了内存分配函数,但是 Cling 的内存分配函数侧重于只复用特定被释放的内存,这段内存的对象类型和释放之前的对象类型一致.Cling 的时间开销与 DangDone 相似,但 Cling 的内存开销略高一些.此外,Cling 不能处理由垂悬指针引起的所有类型的漏洞,例如无法防止由垂悬指针引起的信息泄漏.并且,该方法无法及时检测到出现了 use-after-free 等漏洞,相比之下,DangDone 在预防这类漏洞的同时,也能够通过程序崩溃报告所出现的漏洞.Diehard[27]是一个运行时系统,可以通过随机化和复制来容忍内存错误.可以通过内存随机化和复制来容忍内存错误.它提供了一个近乎无限大的堆空间,使对象可以永远不被释放,并且各个对象可以无限远地被分配.以此消除了垂悬指针,并且不易受到缓冲区溢出攻击.但是空间消耗比传统的内存分配函数大,因为它需要至少两倍的额外内存空间去防止堆被破坏.Dieharder[28]通过改进随机化操作扩展了这种方法,这使得开发变得更困难,并且降低了性能开销.DieHarder 具有 20%的几何平均性能影响,并且该方法可能面临能够控制内存分配的攻击者的攻击.相比之下,内存开销高于 DangDone,少于 DieHarder.

4.2 静态方法

与动态方法或运行时保护相比,检测垂悬指针的静态方法相对少见.大多数静态方法关注一组软件缺陷,垂悬指针只是其中的一个.Musuvati 等人[29]实现一种新的模型检查器,可以直接检查 C/C++程序,无需人工抽象系统行为.自动抽象模型是通过捕获系统的所有行为来实现的[30].因为系统应该是一个面向行为的程序,这种技术的一个局限性是灵活性.Slayer[31]是一个验证工具,用来证明系统代码不存在内存安全错误,例如垂悬指针解引用和 double-free 问题.但是与我们的方法相比,它是一个重量级的工具,因为 slayer 的可扩展性受到 SMT 解算器的限制.

4.3 动态方法

以检测是否出现垂悬指针或其漏洞的方法可以分为两种类型:一种关注 use-after-free 漏洞;另一种是内存错误检测器,它还可以检测垂悬指针或 use-after-free 漏洞.Valgrind Memcheck[32]是一种内存错误检测器,可以检测堆中的 use-after-free 和 double-free 漏洞.与 purify[33]类似,这两种工具都会产生很高的内存和 CPU 开销(代价是 2 倍~25 倍的性能损失),并且无法检测到重新分配数据位置的垂悬指针.Mudflap[34]使用编译工具在指针使用(解引用)时插入判断内存区域是否被合法引用的断言.为了判断内存访问的合法性,它还需要检测来维护有效内存对象的列表.然而,Mudflap 在 SPEC CPU Benchmark 上的额外开销范围从 2x~41x,在复杂的 C++代码中也有误报率.一种类似于 Mudflap 的方法是 AddressSanitizer[35],它是一个基于 Clang 的内存错误检测器.它使用影子内存记录应用程序内存的每个字节是否可以安全访问,并使用检测工具检查每个应用程序加载或存储时的影子内存.但是,为了性能考虑,影子内存没有映射到所有内存分配函数.该优化导致漏报,如果访问重新分配的内存块,将不会检测到 use-after-free.DangDone 会在释放对象时置空垂悬指针,使这些漏洞无法利用.此外,由 AddressSanitizer 所引入的减速是 2 倍,这也高于 DangDone.DangSan[36]方法则是侧重于在多线程下高效率地通过日志检测垂悬指针.它主要优化了记录指针指向、释放信息所需要的时间,以此提高效率.

相比于 RAII 和 C++11 及其之后版本中引入的智能指针(如 *shared_ptr*),这两种方法都能更方便地管理指针.但是相比之下,DangDone 有着更广泛的适用性.这是因为一些系统如 CPS 系统中,不能引入 RAII 或智能指针.除此之外,DangDone 着重于预防和检测垂悬指针,这是 RAII 和智能指针无法直接支持的,需要开发者自行避免垂悬指针及其使用.

4.4 基于程序转换的运行时防御方法

通过程序转换[10,11,37−40]确保空间安全错误的方法与我们的方法类似.为了减少空间安全误差、提高效率,不同的转换策略被提出.以前的方法存在一个或多个不足,很难被广泛采用,例如运行时开销高或需要对现有代码进行很大更改.

DangNull[10],FreeSentry[11]和 pSweeper[39]是最近并且与 DangDone 最相似的预防措施,通过动态运行时检查防止 use-after-free 漏洞.DangNull 由两个主要组件组成:静态检测和运行时库.第 1 个组件通过插入对跟踪例程的调用来标识指针传递,以跟踪指针和内存对象之间的指针指向关系.所有内存操作函数(如 *malloc* 和 *free*)都会被替换,以跟踪内存对象和指针的生命周期.第 2 个组件为静态检测函数提供运行时库,以便动态跟踪指针并将释放的指针及置空其别名.基于 SPEC CPU Benchmark 的评估显示,DangNull 增加了 80%的平均性能开销,这比 DangDone 的开销高得多.FreeSentry 通过将对象链接回其指针来防止 use-after-free 漏洞.在这些链接的帮助下,当对象容易受到垂悬指针的攻击时,FreeSentry 可以使链接到已释放对象的所有指针失效.为了实现这一点,freesentry 注册了创建或修改指向新对象的指针的地址.释放对象后,释放函数将查找该对象所驻留的内存区域的所有指针.如果这些指针仍然指向释放的对象,则将指针置空;如果某些指针试图对失效的内存地址进行解引用,将导致程序崩溃.SPEC CPU Benchmark[13,41]的评估显示:如果禁用栈空间保护,则平均开销为 25%;否则,平均开销为 42%.pSweeper 则是借助另一个线程去扫描已有的指针,判断这些指针是否是垂悬指针,实验表明:pSweeper 的开销可低至 12.5%.

综上所述,可以发现,三者的开销都比 DangDone 高得多.但是 DangDone 依赖于转换规则,而例如 DangNull 等方法的规则则相对更加通用.例如:如果存在嵌套的 struct 结构,DangDone 则还需要增加对应转换规则.本文之前的工作[42]针对所有指针进行防护,但是实际程序它们不一定都是垂悬指针.所以为了减少不必要的转换,我们加入静态分析技术以减少所需要转换的垂悬指针数量.相比之下,前期工作只能够预防垂悬指针而不能通过静态分析检测垂悬指针.

4.5 针对CPS系统的攻击预防措施

CPS 系统在应用层面临的攻击主要通过软件系统漏洞发起的,而在这方面的攻击检测、预防技术主要通过专门模块(如入侵检测模块)实现[43,44].不同于常规软件,CPS 系统可以有独立的模块检测攻击者的恶意输入.如

通过神经网络或支持向量机等分类器,采用数据挖掘、数据融合等方式对不同来源的数据进行分析.利用模式识别方法分析当前输入[43].在攻击检测中,由于恶意输入的特征一般会与常规输入有着较大的差异,该方法也能够在一定程度上检测并预防 use-after-free 或 double free 攻击.第 2 类方法[44]则是侧重于身份认证,该方法基于的前提是攻击者无法获得证明用户身份的证据,如密码、认证证书等.相比于针对 CPS 特征的攻击防御方法,DangDone 则是侧重于在代码层面加固系统本身,避免 CPS 系统内部的缺陷、漏洞暴露出来.

5 总 结

在本文中,我们针对 CPS 系统的垂悬指针问题提出了基于中间指针的垂悬指针预防方法.它先通过静态和动态的方法检测潜在的垂悬指针,再在编译时通过程序转换防止这些指针被利用.我们通过测试 11 个实际漏洞和 5 个 CPS 系统中人工植入的漏洞评估了它在预防 use-after-free 和 double-free 方面的有效性,并使用 SPEC CPU Benchmark 测试评估了运行时开销.实验表明了该方法的有效性和可以忽略不计的运行时开销.在未来,我们计划进一步检测垂悬指针和未被发现的缺陷,如通过模糊测试的方式检测垂悬指针.此外,与大多数现有的预防措施类似,都可能将 use-after-free 和 double-free 问题被转换为空指针解引用问题,这也会影响 CPS 系统的可用性,所以我们计划通过容忍空指针解引用来增强该方法.

References:

[1] Sridhar S, Hahn A, Govindarasu M. Cyber-Physical system security for the electric power grid. Proc. of the IEEE, 2011, 100(1):210−224.

[2] Bu L, Zhang T, Chen X, *et al*. Model-Based construction and verification of cyber-physical systems. ACM SIGSOFT Software Engineering Notes, 2018,43(3):6−10..

[3] Lee E. Cyber physical systems: Design challenges. In: Proc. of the 11th IEEE Int'l Symp. on Object and Component-Oriented Real-Time Distributed Computing (ISORC). IEEE, 2008. 363−369.

[4] Xu W, Li JR, Shu JL, Yang WB, Xie TY, Zhang YY, Gu DW. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2015. 414−425.

[5] Ye JY, Zhang C, Han XH. POSTER: UAFChecker: Scalable static detection of use-after-free vulnerabilities. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security. 2014. 1529−1531.

[6] Feist J, Mounier L, Potet ML. Statically detecting use after free on binary code. Journal of Computer Virology and Hacking Techniques, 2014,10(3):211−217.

[7] Caballero J, Grieco G, Marron M, Nappa A. Un-Dangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proc. of the Int'l Symp. on Software Testing and Analysis. ACM, 2012. 133−143.

[8] Dhurjati D, Adve V. Efficiently detecting all dangling pointer uses in production servers. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. IEEE, 2006. 269−280.

[9] Pattabiraman K, Kalbarczyk ZT, Iyer RK. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. IEEE Trans. on Dependable and Secure Computing, 2011,8(1):44−57.

[10] Lee BY, Song CY, Jang YJ, Wang TL, Kim TS, Lu L, Lee WK. Preventing use-after-free with dangling pointers nullification. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2015.

[11] Younan Y. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2015.

[12] The LLVM Compiler Infrastructure. 2020. http://llvm.org

[13] Standard Performance Evaluation Corporation. SPEC CPU 2006. 2020. https://www.spec.org/cpu2006/

[14] Afek J, Sharabani A. Dangling Pointer: Smashing the Pointer for Fun and Profit. Black Hat USA, 2007. 24.

[15] Jackson T, Salamat B, Wagner G, Wimmer C, Franz M. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In: Proc. of the 6th Int'l Workshop on Security Measurements and Metrics. ACM, 2010.

[16] Lopes BC, Auler R. Getting Started with LLVM Core Libraries. Packt Publishing Ltd, 2014.

[17] Steensgaard B. Points-to analysis in almost linear time. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 1996. 32−41.

[18] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. København: University of Cophenhagen, 1994.

[19] Szekeres L, Payer M, Wei T, Song D. Sok: Eternal war in memory. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2013. 48−62.

[20] Writing an LLVM Pass. 2020. http://llvm.org/docs/WritingAnLLVMPass.html

[21] Clang: A C language Family Frontend for LLVM. 2020. http://clang.llvm.org/

[22] Exploits Database by Offensive Security. Exploits database. 2020. https://www.exploit-db.com/

[23] Fortify Static Code Analysis Tool. 2020. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[24] Splint-Secure Programming Lint. 2020. http://www.splint.org/

[25] Coverity Scan-static Analysis. 2020. https://scan.coverity.com/

[26] Akritidis P. Cling: A memory allocator to mitigate dangling pointers. In: Proc. of the USENIX Security Symp. 2010. 177−192.

[27] Berger ED, Zorn BG. DieHard: Probabilistic memory safety for unsafe languages. ACM SIGPLAN Notices, 2006,41(6):158−168.

[28] Novark G, Berger ED. DieHarder: Securing the heap. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. ACM, 2010. 573−584.

[29] Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. ACM SIGOPS Operating Systems Review, 2002,36(SI):75−88.

[30] Engler D, Musuvathi M. Static analysis versus software model checking for bug finding. In: Proc. of the Verification, Model Checking, and Abstract Interpretation. Springer-Verlag, 2004. 191−210.

[31] Berdine J, Cook B, Ishtiaq S. SLAyer: Memory safety for systems-level code. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2011. 178−183.

[32] Morgado A. Understanding valgrind memory leak reports. 2020. https://aleksander.es/data/valgrind-memcheck.pdf

[33] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. In: Proc. of the Winter 1992 Usenix Conf. Citeseer, 1991. 125−136.

[34] Eigler FC. Mudflap: Pointer use checking for C/C+. In: Proc. of the 1st Annual GCC Developers' Summit. 2003. 57−70.

[35] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A fast address sanity checker. In: Proc. of the USENIX Annual Technical Conf. 2012. 309−318.

[36] Van Der Kouwe E, Nigade V, Giuffrida C. Dangsan: Scalable use-after-free detection. In: Proc. of the 12th European Conf. on Computer Systems, ACM, 2017. 405−419.

[37] Nagarakatte S, Zhao JZ, Martin MMK, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. In: Proc. of the ACM Sigplan Notices. 2009. 245−258.

[38] Xu W, Duvarney DC, Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In: Proc. of the ACM Sigsoft Int'l Symp. on Foundations of Software Engineering. ACM, 2004. 117−126.

[39] Liu DP, Zhang MW, Wang HN. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2018. 1635−1648.

[40] Shin JS, Kwon DH, Seo JW, Cho YP, Paek YH. CRCount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2019.

[41] Standard Performance Evaluation Corporation. SPEC CPU 2000. 2020. https://www.spec.org/cpu2000/

[42] Wang Y, Gao FJ, Situ LY, Wang LZ, Chen BH, Liu Y, Zhao JH, Li XD. Dangdone: Eliminating dangling pointers via intermediate pointers. In: Proc. of the 10th Asia-Pacific Symp. on Internetware. ACM, 2018. 6.

[43] Su CJ, Wu CY. JADE implemented mobile multi-agent based, distributed information platform for pervasive health care monitoring. Applied Soft Computing, 2011,11(1):315−325.

[44] Liang HY, Jagielski M, Zheng BW, *et al.* Network and system level security in connected vehicle applications. In: Proc. of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD). IEEE, 2018.

王豫(1991－),男,浙江献县人,学士,主要研究领域为软件工程,程序语言,软件安全.

高凤娟(1991－),女,学士,主要研究领域为软件工程,程序语言,软件安全.

马可欣(1999－),女,本科生,主要研究领域为软件工程.

司徒凌云(1988－),男,硕士,CCF 学生会员,主要研究领域为软件工程,软件安全.

王林章(1973－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件工程,软件安全.

陈碧欢(1986－),男,博士,CCF 专业会员,主要研究领域为软件工程,软件安全.

刘杨(1981－),男,博士,教授,博士生导师,主要研究领域为软件工程,网络空间安全,人工智能.

赵建华(1971－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程.

李宣东(1963－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为软件工程.

www.jos.org.cn