软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(5):1179−1198 [doi: 10.13328/j.cnki.jos.005504]

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

# 缓冲区溢出漏洞分析技术研究进展*

邵思豪[1,2], 高 庆[1,3], 马 森[1,2], 段富尧[1,2], 马 骁[1,2], 张世琨[1,2], 胡津华[4]

[1](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[2](软件工程国家工程研究中心(北京大学),北京 100871)

[3](北京大学 信息科学技术学院,北京 100871)

[4](天津市职业病防治院(天津市工人医院) 信息科,天津 300011)

通讯作者: 高庆, E-mail: gaoqing@pku.edu.cn; 马森, E-mail: masen@pku.edu.cn

**摘 要**: 首先介绍了缓冲区溢出漏洞危害的严重性和广泛性,然后,从如何利用缓冲区溢出漏洞的角度,依次介绍了缓冲区溢出漏洞的定义、操作系统内存组织方式以及缓冲区溢出攻击方式.将缓冲区溢出分析技术分为 3 类:自动检测、自动修复以及运行时防护,并对每一类技术进行了介绍、分析和讨论.最后,对相关工作进行了总结,并讨论了缓冲区溢出分析领域未来可能的 3 个研究方向:(1) 对二进制代码进行分析;(2) 结合机器学习算法进行分析;(3) 综合利用多种技术进行分析.

**关键词**: 缓冲区溢出漏洞;攻击;分析;误报率;漏报率

**中图法分类号**: TP311

中文引用格式: 邵思豪,高庆,马森,段富尧,马骁,张世琨,胡津华.缓冲区溢出漏洞分析技术研究进展.软件学报,2018,29(5):1179−1198. http://www.jos.org.cn/1000-9825/5504.htm

英文引用格式: Shao SH, Gao Q, Ma S, Duan FY, Ma X, Zhang SK, Hu JH. Progress in research on buffer overflow vulnerability analysis technologies. Ruan Jian Xue Bao/Journal of Software, 2018,29(5):1179−1198 (in Chinese). http://www.jos.org.cn/1000-9825/5504.htm

## Progress in Research on Buffer Overflow Vulnerability Analysis Technologies

SHAO Si-Hao[1,2], GAO Qing[1,3], MA Sen[1,2], DUAN Fu-Yao[1,2], MA Xiao[1,2], ZHANG Shi-Kun[1,2], HU Jin-Hua[4]

[1](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[2](National Engineering Research Center for Software Engineering (Peking University), Beijing 100871, China)

[3](School of Electrical Engineering and Computer Science, Peking University, Beijing 100871, China)

[4](Information Department, Tianjin Occupational Diseases Precaution and Therapeutic Hospital (Tianjin Workers' Hospital), Tianjin 300011, China)

**Abstract**: First, in this paper, the breadth and risk of buffer overflow vulnerabilities are introduced. Then, from the aspect of how to exploit a buffer overflow vulnerability, an overview is provided on the definition of buffer overflow vulnerabilities, memory organization in operation systems, and classification of buffer overflow attacks. Based on the research, buffer overflow analysis technologies are classified into three categories: automatic detection, automatic repair, and run-time protection. Each types of technologies are introduced, analyzed and discussed according to the classification. Finally, three possible research directions in the field of buffer overflow vulnerability analysis are discussed: (1) analyzing binary code; (2) using machine learning algorithms; (3) combining multiple technologies for analysis.

* 基金项目: 国家重点研发计划(2017YFB0802900); 中国博士后科学基金(2017M620524); 北京市自然科学基金(4182024)

Foundation item: National Key Research and Development Program of China (2017YFB0802900); China Postdoctoral Science Foundation (2017M620524); Beijing Municipal Natural Science Foundation (4182024)

本文由软件安全漏洞检测专题特约编辑王林章教授、陈恺研究员、王戟教授推荐.

收稿时间: 2017-07-02; 修改时间: 2017-08-29; 采用时间: 2017-11-21; jos 在线出版时间: 2018-01-09

**Key words**: buffer overflow vulnerability; attack; analysis; false positive rate; false negative rate

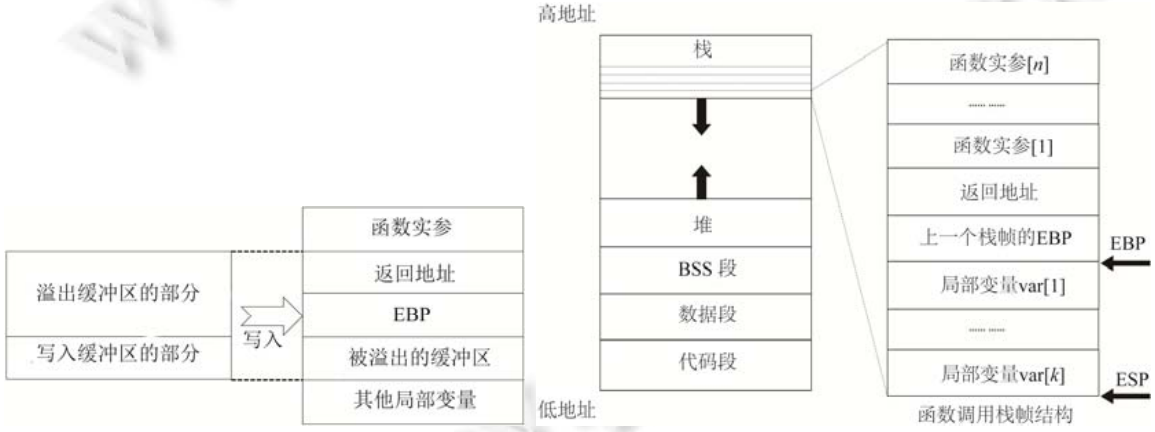
自 1988 年由罗伯特•莫里斯制造的 Morris 蠕虫病毒利用缓冲区溢出漏洞造成全世界 6 000 多台网络服务器瘫痪以来,缓冲区溢出漏洞的广泛性和危险性越来越受到国内外信息安全研究领域的关注.在 2004 年 OWASP 的 Web 应用十大最严重的安全风险中,缓冲区溢出漏洞排名第五[1].在 2010 年和 2011 年 CWE/SANS 机构总结的危害性最强的 25 大软件缺陷中,缓冲区溢出漏洞都名列第三[2];同时,CWE 自身的软件脆弱性枚举库中有 21 条和缓冲区溢出漏洞相关的条目.在中国科学院大学国家计算机网络入侵防范中心总结的 2014 年 11 月十大重要安全漏洞分析中,缓冲区溢出漏洞排名第二[3].国家信息安全漏洞库(CNNVD)报告显示:2015 年,国内信息技术产品新增漏洞 7 754 个,其中,缓冲区溢出漏洞 1 088 个,占比最高为 14.03%[4].尽管国内外科研人员针对缓冲区溢出漏洞广泛存在于软件中这一棘手问题提出了许多分析技术和方法,然而,这一问题至今也没有得到完全的解决.南京大学的叶涛、王林章、李宣东等人[5]在 2016 年的研究结果显示,目前的分析技术对 CVE[6] 中的缓冲区溢出漏洞的检测及修复能力仍差强人意.因此,对近年来的缓冲区溢出漏洞分析技术进行梳理总结、分析探讨,以辅助科研人员、工程技术人员更好地进行研究和应用,是非常有必要的.

本文第 1 节从如何利用缓冲区溢出漏洞的角度,简要地介绍缓冲区溢出攻击.第 2 节根据调研结果对缓冲区溢出分析技术进行分类,在分类的基础上对每一类技术进行介绍.第 3 节本文对比分析上述三大类分析技术.第 4 节对本文所做工作进行总结,并提出未来可能的研究热点.

## 1 缓冲区溢出攻击简介

### 1.1 缓冲区溢出漏洞的定义

缓冲区是操作系统内存中一段连续的存储空间.当一段程序尝试把更多的数据放入一个缓冲区,数据超出缓冲区的预留范围时,或者说当一段程序尝试把数据放入的内存位置超出了缓冲区的边界时,便触发了缓冲区溢出漏洞[2].如图 1(a)所示,如果向一个缓冲区写入数据,并且写入的数据量比缓冲区大时,缓冲区溢出漏洞就会被触发[7].缓冲区溢出漏洞使攻击者能够执行恶意代码或者使程序崩溃[8].由于广泛使用的 C 和 C++程序语言没有对数组读写数据进行边界检查的机制,导致了这一漏洞常常被攻击者所利用[9].




(a) 缓冲区溢出原理 (b) Linux 系统内存组织方式

Fig.1 Principle of buffer overflow and memory organization in Linux

图 1 缓冲区溢出原理和 Linux 系统内存组织方式

### 1.2 操作系统内存组织方式

为了更准确地理解缓冲区溢出漏洞,我们有必要了解操作系统中内存的组织方式.操作系统内存通常由 4 部分组成.

1) 代码段:存放可执行文件的操作指令.
2) 数据段:存放可执行文件中的全局变量和静态变量.
3) 堆:存放程序运行过程中动态分配的内存.
4) 栈:存放一些临时数据.在某个函数被调用时,栈中依次压入 ARG(函数调用时的实参)、RETADDR(下一条要执行的操作指令在内存中的地址)、EBP(上一个栈帧的 EBP 值)和 LOCVAR(该函数中的局部变量).在一次函数调用完成后,这些内容又从栈中移除.

以图 1(b)所示的 Linux 系统中内存的组织方式为例,从内存低端到内存高端依次是:代码段、数据段(存放可执行文件中已初始化的全局变量和静态变量)、BSS 段(存放可执行文件中未初始化的全局变量和静态变量)、堆和栈.Linux 系统内存中的数据段和 BSS 段两部分组合起来,可以看作操作系统中广义的数据段部分.

### 1.3 缓冲区溢出攻击方式

缓冲区溢出攻击按照攻击后果的严重程度可分为导致程序崩溃(多数情况下会导致拒绝服务攻击)和获得系统控制权限两种.通常,缓冲区溢出攻击都是按照如下步骤进行:(1) 注入攻击代码;(2) 跳转到攻击代码;(3) 执行攻击代码.其中,第(2)步是利用缓冲区溢出漏洞实现攻击的核心环节[10].

缓冲区溢出按照所攻击对象的不同可分为 3 类.

(a) 破坏栈数据

栈数据中的 ARG(函数调用时的实参)、RETADDR(下一条要执行的操作指令在内存中的地址)、EBP(调用该函数前的栈帧状态值)和 LOCVAR(该函数中的本地变量)都是可能被攻击者利用缓冲区溢出漏洞进行破坏的对象.其中最常见的就是利用缓冲区溢出改变 RETADDR 的值,使其存放已经注入到栈中的攻击代码的地址或者是代码区中某些具有特权的系统函数地址(比如 system).如果修改成功,当该函数调用完毕后,程序就跳转到攻击者设计好的地址去执行攻击者希望被执行的指令,进而获得系统控制权限,导致严重的后果.

此外,EBP 也常常作为被攻击的对象[11].攻击者通过构建一个 RETADDR 指向攻击代码的虚拟栈帧,再溢出当前栈帧 EBP 的值,让溢出后的 EBP 值是构造的虚拟栈帧的地址.这样,最终通过构造的虚拟堆栈的承接,执行完当前栈帧则执行虚拟栈帧,执行完虚拟栈帧则跳转到虚拟栈帧的 RETADDR 值所指向的位置,也使得程序最终跳转到攻击者设计好的地址去执行攻击指令.

(b) 破坏堆数据

由于堆中是不连续地动态分配内存,攻击者预测地址的难度提高,堆溢出攻击比栈溢出攻击更困难,但依然有技术可以利用堆溢出实现攻击.

- Dword Shoot 攻击:Linux 系统和 windows 系统对堆的管理方式都是双向链表方式[12],每一个分配的内存块由 3 部分组成:头指针(head)、尾指针(tail)、内存数据(data).针对堆内存的管理主要有分配和释放两部分.在释放堆内存 $M$ 时,会将 $M$ 从链表上摘除,会执行 $M\rightarrow head\rightarrow tail=M\rightarrow tail$ 操作,如果攻击者通过溢出 $M$ 临近的内存,将 $M$ 的头指针、尾指针修改,让 $M$ 的头指针指向攻击者设计好的虚拟节点,让 $M$ 的尾指针指向攻击者设计好的位置,比如 Shellcode,那么当执行完 $M\rightarrow head\rightarrow tail=M\rightarrow tail$ 操作时,该虚拟节点的尾指针就指向 Shellcode,调用该虚拟节点的尾指针就会转向 Shellcode.摘链时的另一操作 $M\rightarrow tail\rightarrow head=M\rightarrow head$,利用同样的原理,也可实现攻击.
- Heap Spray 攻击:攻击者向系统申请大量内存,并将有大量滑板指令的 Shellcode 作为注入攻击载荷(所谓滑板指令是指不会影响程序执行,但却占据内存空间,因此有利于真正的攻击指令得到执行的无意义指令),反复进行填充.然后结合 ROP 等技术控制程序流,使得程序跳转执行到被大量注入的攻击载荷所占据的堆上,从而 Shellcode 得到执行,Shellcode 中的滑板指令不会影响程序执行,Shellcode 中的核心攻击指令得到执行,进而获得系统控制权限,达到攻击目的.
- Heap fengshui 攻击:对于 Windows XP SP2 之后的微软的操作系统来说,由于堆中内存空间的分配函数(HeapAlloc)在其分配的内存块(chunk)的管理头部中加入了 cookie 校验信息,并且实现了空闲内存块的双向链表的安全删除技术,因此利用堆腐烂技术覆盖内存块的管理头部中的数据变得非常困难[13].

另外,Heapspray 技术需要占用大量系统资源,且可能耗时较长,针对上述问题,Sotirov[13]提出了 Heap fengshui 攻击技术,该技术根据操作系统堆内存管理方式,通过主动设计回收内存、清理堆中内存碎片来保证攻击者分配的内存在物理上是连续的.与 Heap Spray 攻击相比,该技术无需反复填充攻击代码,占用内存较为合理,实用价值较高.

(c) 更改类函数指针

类函数指针分为两类:一类是真正意义上的函数指针,它指向函数的存储位置,可通过解引用指针的方式调用函数;另一类是固定缓存区(*jmp_buf* 类型变量),该类型变量存放程序的状态信息,用于实现 C 语言中的非局部跳转.固定缓存区的正确用法有如下两步:首先,通过 *setjmp*()宏把当前程序的状态信息保存到 *jmp_buf* 类型变量中;然后,通过 *longjmp*()恢复程序状态信息进而实现非局部跳转.

固定缓存区和函数指针功能类似,前者存放的是程序的状态信息,后者存放的是函数的位置信息.它们都能实现跳转到某位置,然后去执行某段指令的功能,这为攻击者提供了机会.攻击者通过溢出类函数指针附近的缓冲区,以达到改写类函数指针中存放内容的目的,进而使程序跳转到攻击者设计好的位置,去执行攻击者希望被执行的代码,以达到获得系统控制权限或者导致程序崩溃的目的.更改类函数指针可以攻击内存中栈、堆、数据段.当程序采用防止 RETADDR 被修改的策略进行保护后,这种攻击方式比较有效,因为它不会更改保存的 RETADDR[14].

## 2 分析技术分类

缓冲区溢出漏洞分析技术旨在防止软件内部的缓冲区溢出漏洞被外部攻击者利用[15].因此,缓冲区溢出漏洞分析技术可以按照应用场景分为漏洞自动检测技术、漏洞自动修复技术和漏洞运行时防护技术.

- 漏洞自动检测技术:通过静态检测、动态测试的方式,在软件发布前,及时地发现软件中的缓冲区溢出漏洞,以便于对源代码进行人工修改.
- 漏洞自动修复技术:在检测出软件中的缓冲区溢出漏洞后,通过先进的分析手段,自动修改源代码或二进制代码,将软件中的缓冲区溢出漏洞剔除掉,对其进行修复.
- 漏洞运行时防护技术:在软件发布后,不再对源代码进行更改,站在防止攻击者利用的立场,对某些关键对象/性质进行完整性/机密性/可用性保护(如对 RETADDR 进行完整性保护).相当于把可能存在缓冲区溢出漏洞的软件放在一个安全的保护罩里去运行.

下分别对上述三大类技术做简要介绍.

### 2.1 漏洞自动检测技术

如第 1.2 节所述的内存组织方式,一旦发生堆缓冲区溢出或栈缓冲区溢出,将会使数据越过相应的内存块,覆盖堆或栈的其他字节,甚至写到函数返回地址及数据区,从而带来严重后果.因此,很多研究关注自动化地发现软件漏洞,通过检测缓冲区访问索引与缓冲区大小之间的关系来判定缓冲区溢出的发生.具体分为静态检测和动态测试两种方法.

#### 2.1.1 静态检测

静态检测是指在不运行软件前提下进行的检测过程[16].静态检测的对象一般是程序源代码,也可以是目标码[16].静态检测多数情况下可以准确地定位到程序所在位置,便于编程人员修改,从而彻底将检测出的漏洞消除.但由于静态检测缺少程序运行时信息,因此很难做到误报率和漏报率都很低.除了误报率和漏报率以外,吞吐量也是衡量一种静态检测技术是否有效、实用的重要指标.

静态检测技术采用的分析技术是静态分析技术.静态分析按照对模型节点的遍历方法可以分为正向分析、逆向分析、无向分析.正向分析是指从可能发生缓冲区溢出漏洞的源节点出发,正向遍历模型中的节点,在遍历的过程中生成相应约束或提取部分属性进行分析的分析方法;逆向分析是指从可能发生缓冲区溢出漏洞的槽(sink)节点出发,逆向遍历模型中的节点,在遍历的过程中生成相应约束或提取部分属性进行分析的分析方法;无向分析是指在分析过程中不考虑节点的前驱节点、后继节点和到达节点的条件等语义信息,只考虑节点自身

的语法信息进行分析的方法.

缓冲区溢出漏洞静态检测技术可以按照主要采用的技术种类、采用技术的深度及静态分析的侧重点分为:(1) 基于抽象解释的缓冲区溢出漏洞静态检测技术;(2) 基于符号执行的缓冲区溢出漏洞静态检测技术;(3) 基于污染传播的缓冲区溢出漏洞静态检测技术;(4) 基于特征分类的缓冲区溢出漏洞静态检测技术.而约束的生成与求解、模式匹配、图可达分析、数据流分析等技术常常作为共用技术辅助缓冲区溢出漏洞静态检测.下面分别介绍上述 4 种技术的典型代表.

- 基于抽象解释的缓冲区溢出漏洞静态检测技术

Wagner 等人[17]将缓冲区溢出漏洞的检测抽象为整数约束生成与求解问题.首先,为每个缓冲区增加两个属性:一个是 Size(缓冲区分配大小的值域范围),另一个是 Length(缓冲区访问大小的值域范围);然后,对每一个字符串操作库函数进行字符串运算定义;接下来,在每一个访问缓冲区溢出的位置生成相应的约束(*Size*>*Length*);最后进行约束求解,若约束条件有可能不被满足,则生成警告.

Larochelle 等人[18]借助开源标记注解工具 LCLint 对每一个可能发生缓冲区溢出漏洞的函数调用点前后进行标记,当调用该函数时,判断约束是否可以满足.该方法对循环的处理采用了启发式算法,该方法认为:对于一个循环,初值和终值最有可能导致缓冲区溢出,因此该方法只关注循环的初值和终值.该方法的优点是吞吐量在当时技术条件下较大(相比 Wagner 等人的方法),缺点是误报率和漏报率都较高.

宫云战等人在文献[19–21]中引入了区间运算对程序中出现变量的取值范围进行评估,同时对 C/C++字符串操作、内存操作函数进行函数摘要的计算.该方法在函数调用点根据区间信息和函数摘要技术进行比较判断是否存在缓冲区溢出漏洞.由于上下文敏感分析的时空开销较大,该方法没有采用上下文敏感的分析.虽然在构建函数摘要过程中,利用状态机对字符串操作这类函数分析得更精准,但由于区间分析的不完备性和该方法不是上下文敏感的分析,导致存在一定的误报.另外,该方法只对字符串操作、内存操作函数涉及到的缓冲区进行判断,导致一定的漏报.但该方法的吞吐量较大、实用性较好,以上述论文为原型研发的缓冲区溢出检测工具是国内第一套缓冲区溢出静态分析工具.

来自于 NASA 的 Brat 等人[22]针对航天航空的认证需求——航天航空软件通常代码行数巨大,且要求分析尽可能是全面而准确的,采用抽象解释的方法构造了吞吐量极大、误报率较低,但空间开销较大、分析速度较慢的缓冲区溢出检测方法.该方法对 LLVM[23]框架进行了一定的改动,设计了适应其需求的中间语言 AR,AR 相对于 LLVM 的 IR 具有指令集更规则、去掉 SSA 表达方式、以声明方式表达选择语句等特点,方便采用抽象解释方法进行分析.

- 基于符号执行的缓冲区溢出漏洞静态检测技术

Le 等人[24,25]开发的检测工具 Marple 把控制流图作为其分析的基础模型,并把控制流图中的路径分为不可达、安全、漏洞、警告、不知道这 5 类.该方法的分析过程如下.

(1) 首先对程序中的路径进行分析,区分开可达路径与不可达路径,并根据漏洞模型找出有潜在溢出风险的语句.
(2) 接下来,依靠漏洞模型中的安全规则去生成查询,该安全规则是确保缓冲区溢出不被触发的形式化约束条件,例如字符数组的大小应大于对其访问的索引下标值.
(3) 查询生成后,从它被提出的位置沿着可达的路径向着 entry 节点逆向传播.
(4) 传播过程中,依据 marple 内部的规则对查询进行更新.
(5) 并且,当下面两个条件满足其一时,传播即被终止:a) 当前传播到的节点前驱节点是不可达节点或者 entry 节点;b) 查询在更新后可求解.
(6) 最后,根据查询的可满足情况进行漏洞判断,如果被判定是漏洞或者警告,则将路径类型信息、缓冲区溢出根源信息、漏洞具体路径信息报告给用户[24].

Marple 是路径敏感、流敏感的分析,并且上下文敏感,因此误报率极低.但是其性能开销大,并且由于其把很多路径归为“不知道”,导致有一定的漏报.

Ding 等人[26]针对 Marple 吞吐量较低并且漏报率较高的问题,提出了基于模式识别的限制性的符号执行检测方法.首先,其总结了 3 种导致控制流分支中存在缓冲区漏洞的代码模式:脆弱的语法用法模式、脆弱的元素访问模式和脆弱的成批移动模式;然后,使用语法分析过滤掉第 1 种模式,并只发送符合第 2 种、第 3 种模式的节点和分支来进行符号评估,这也是该方法限制性符号执行的体现[26].由于第 1 种模式涉及到很多节点,该方法针对符合第 1 种模式的节点的处理方法比 Marple 简单很多,因此节约了很多时空开销,吞吐量更大.但又因为在实际应用程序中,有部分分支虽然符合第 1 种模式,但实际输入会避免那些不安全的操作,所以该分支事实上是安全的,这导致该方法产生了部分误报.此外,该方法将循环抽象成基本块,而 Marple 每次都是用一个固定的次数迭代循环,这使得该方法在循环的处理方面比 Marple 分析得更准确[26].

- 基于污染传播的缓冲区溢出漏洞静态检测技术

Gao 等人[27]针对缓冲区溢出漏洞中最常见的一类缺陷——数组下标越界缺陷,提出了基于控制流图和函数调用图,执行污染传播分析和数据流分析的检测方法,并实现了原型工具 CarrayBound.首先,Carraybound 执行污点分析来确定所有变量的污染传播状态;然后,Carraybound 找到包含数组表达式的所有语句,并构造数组边界信息;接下来,通过后向数据流分析遍历控制流图,去验证是否存在语句能够确保数组下标在数组边界范围内[27],在验证这一步,先是函数内分析,若无法确定则进行跨函数分析;最后,若不存在能够确保数组下标在数组边界范围内的语句,则将这些缺陷报告出来.该工具 10min 内测出了 250 千行以上的 PHP-5.6.16 工程中多个有价值的数组越界缺陷.这证明了该技术具有吞吐量大、测试速度快、针对数组越界这类缺陷漏报率低的特点.不足的是,该技术仅针对数组下标越界缺陷,检测的缺陷模式不够全面.

- 基于特征分类的缓冲区溢出漏洞静态检测技术

Bindu 等人[28]将静态分析和机器学习算法结合起来检测缓冲区溢出漏洞.首先,分别对槽节点、输入类型、输入验证节点及缓冲区大小判断节点类型、槽节点属性这 4 个代码属性进行分类.其定义了 7 种类型的槽节点、4 种输入类型、13 种输入验证节点及缓冲区大小判断节点类型、9 类槽节点属性;然后,用静态分析中常用的 TP,FP,TN,FN 作为衡量指标,分别采用朴素贝叶斯、决策树、多层感知机、逻辑回归、支持向量机这 5 种分类器进行机器学习;学习完成后,再用 5 个基准测试程序进行测试,验证工具的检测能力.因为该方法关注的代码属性都是很容易就能收集到的,并且该算法没有像常规的静态分析算法采用符号执行或者约束求解的方法,因此算法很高效,吞吐量很大.这 5 个基准测试程序的测试结果显示,该工具的误报率和漏报率都极低.但仅测试 5 个基准程序无法证明该工具在误报率和漏报率方面一定优于其他技术.

Bindu 等人[29]采用 IDA Pro 反汇编工具对二进制文件进行反汇编,然后采用类似的技术对汇编语言进行缓冲区溢出检测.该技术最大的特点是实现了对二进制文件的检测.

表 1 是上述静态检测方法的对比,其中,指标标准见表 2.

**Table 1** Comparison of static detection approaches

**表 1** 静态检测方法对比

| 方法分类 | | 抽象解释 | | | | 符号执行 | | 特征分类 | | 污染传播 |
|---|---|---|---|---|---|---|---|---|---|---|
| 典型代表 | | Wagner 等人 (2000) | Larochelle 等人 (2001) | 宫云战等人 (2012) | Brat 等人 (2014) | Le 等人 (2008) | Ding 等人 (2012) | Bindu 等人 (2014) | Bindu 等人 (2015) | Gao 等人 (2016) |
| 技术特点 | 结点遍历方法 | 正向分析 | 正向分析 | 正向分析 | 正向分析 | 逆向分析 | 逆向分析、无向分析 | 正向分析 | 正向分析 | 正向分析、逆向分析 |
| | 循环处理 | 求解循环不动点 | 只关注循环的初值和终值 | 求解循环不动点 | 求解循环不动点 | 用一个固定次数迭代循环 | 将循环抽象成一个基本块 | 无 | 无 | 求解循环不动点 |

**Table 1** Comparison of static detection approaches (Contiuned)

**表 1** 静态检测方法对比(续)

| 方法分类 | | 抽象解释 | | | | 符号执行 | | 特征分类 | | 污染传播 |
|---|---|---|---|---|---|---|---|---|---|---|
| 典型代表 | | Wagner 等人(2000) | Larochelle 等人(2001) | 宫云战等人(2012) | Brat 等人(2014) | Le 等人(2008) | Ding 等人(2012) | Bindu 等人(2014) | Bindu 等人(2015) | Gao 等人(2016) |
| 技术特点 | 判别方法 | 抽象解释约束求解 | 抽象解释约束求解 | 区间分析、函数摘要 | 抽象解释 | 符号执行约束求解 | 有限符号执行、语法分析、约束求解 | 机器学习 | 机器学习 | 污染传播分析、数据流分析 |
| 分析敏感性 | 路径 | 否 | 是 | 是 | 是 | 是 | 部分 | 否 | 否 | 是 |
| | 流 | 否 | 是 | 是 | 是 | 是 | 部分 | 是 | 是 | 是 |
| | 上下文 | 否 | 否 | 否 | 否 | 是 | 部分 | 否 | 否 | 否 |
| 主要优点 | | 吞吐量与现有技术相比虽然较小,但在当时较大 | 吞吐量与现有技术相比虽然较小,但比 Wagner 的方法大 | 1. 吞吐量大<br>2. 对字符串操作函数导致的缓冲区溢出检测漏报率较低 | 1. 吞吐量大<br>2. 误报率极低<br>3. 漏报率较低 | 误报率极低 | 吞吐量大、漏报率较低 | 1. 性能开销较小<br>2. 吞吐量较大 | 1. 性能开销较小<br>2. 吞吐量较大<br>3. 能够分析二进制文件 | 1. 对数组下标越界这类缺陷检测效果较好<br>2. 性能开销较小 |
| 主要缺点 | | 误报率极高 | 误报率、漏报率都较高 | 对非字符串操作函数导致的缓冲区溢出检测漏报率较高 | 空间开销大 | 性能开销高 | 误报率较高 | 测试工程较少,无法保证有普适性、在大量实际工程检测时误报率很有可能较高 | 测试工程较少,无法保证有普适性、在大量实际工程检测时误报率很有可能较高 | 缺陷模式不够全面 |

**Table 2** Index standard

**表 2** 指标标准

| 指标标准 | 极低/极小 | 较低/较小 | 高/大 | 较高/较大 | 极高/极大 |
|---|---|---|---|---|---|
| 误报率(%) | ≤10 | 10~20 | 20~50 | 50~80 | ≥80 |
| 漏报率(%) | ≤10 | 10~20 | 20~50 | 50~80 | ≥80 |
| 吞吐量(千行) | 0~10 | 10~100 | 100~500 | 500~1 000 | ≥1 000 |
| 性能开销(GB) | ≤1 | 1~2 | 2~8 | 8~16 | 16~32 |

### 2.1.2 动态测试

动态测试技术是指从通常无限大的执行域中恰当地选取一组有限的测试用例来运行程序,从而检验程序的实际运行结果是否符合预期结果的分析手段[30].基于动态测试的缓冲区溢出检测工具需要在检测对象编译生成的目标码中置入动态检测代码或断言的基础上运行测试用例,观察待测程序.该方法能够在一定程度上检测出缓冲区溢出漏洞,但是生成及运行测试用例时性能开销较大.并且由于无法做到测试用例完全覆盖程序中所有可执行路径,有漏报率较高的缺点.动态测试技术的核心在于如何生成覆盖率高的测试用例,或者生成虽然覆盖率不高,但是能够命中要害、触发缓冲区溢出漏洞的发生的测试用例.如何高效地产生能够到达并触发应用程序漏洞部分的测试用例,是该技术的最大挑战[31].

缓冲区溢出漏洞动态测试技术可按照静态分析技术的参与程度分为 3 种.

1) 不利用静态分析:在整个测试过程中都没有利用静态分析技术分析程序源代码,例如黑盒测试等.
2) 静态分析辅助输入:在测试过程中,静态分析辅助动态测试产生触发溢出漏洞的测试用例,但静态分析本身并不作为独立的检测部分去生成漏洞结果.
3) 静态分析协同检测:在测试过程中,有独立的静态检测部分产生部分溢出漏洞结果,与动态检测组合

起来,综合二者优点进行溢出漏洞检测,以求覆盖率更高、更全面地发现程序中的漏洞.

下面分别介绍上述 3 种技术的典型代表.

- 不利用静态分析的缓冲区溢出漏洞动态测试技术

Wang 等人[32]采用组合测试的测试用例生成技术,没有使用符号执行技术和遗传算法.首先,找到程序中所有的外部输入变量的集合 *P*,并将 *P* 分为 3 个部分:载荷攻击参数(attack-payload parameter)集合 *Px*、攻击控制参数(attack-control parameter)集合 *Pc* 和非攻击控制参数(non-attack-control parameter)集合 *Pd*,并且将 *Px* 固定赋值为一个很长的字符串,将 *Pc* 赋值为 0.接下来,使用 ACTS 方法[33]对 *Pc* 生成 *t* 路测试集,保证对 *Pc* 集合任意 *t* 个参数做到分支路径全覆盖.该方法成功地将测试用例个数降到了 $O(\max(|Px|,|Pc|)^{(t+1)}\times|P|\times\log|P|)$[32].虽然该方法能够在保证较高的测试覆盖率的同时进行测试用例的消减,但是测试用例数量级并没有改变,时空开销依然较高.在超过 10 万行的工程中的效率并不令人满意,并且该方法只考虑了没有被限制长度的输入字符串导致的缓冲区溢出,有一定的使用局限性.

- 静态分析辅助输入的缓冲区溢出漏洞动态测试技术

Haller 等人[34]综合使用污染数据传播分析、数据流分析和符号执行等技术来选择合适的位置插桩和生成恰当的测试用例.他们认为:循环里的数组访问由于循环的复杂性,程序行为很有可能超出编程人员的预期,是最有可能发生缓冲区溢出漏洞的节点.所以先将循环中对数组访问的节点作为候选节点集合,再根据评估模型对这些候选节点进行评分(分数越高,代表越有可能发生溢出,例如,某个缓冲区指针在解引用前进行算术加减,那么该指针解引用对应的缓冲区访问节点得分就会较高),对于得分较高(发生溢出可能性较大)的节点,利用相对成熟的污染数据传播技术找到所有可能影响到数组下标值的外界输入,并将其符号化,沿着执行路径进行传播.传播完毕后,得到与漏洞相关的输入变量构成的约束,基于该约束生成相对应的测试用例.最后,借助第三方插件用生成的测试用例测试缓冲区上溢出和下溢出.该方法的优点是生成的测试用例比较有针对性,对于由循环里的数组访问导致的缓冲区溢出漏洞测试效果很好.缺点是只考虑了这一类缓冲区溢出,并且选择待测试节点这一步骤采用了多项技术导致该步骤时空开销较大.

Bindu 等人[35]采用的测试用例生成算法比较简单,既没有使用符号执行技术,也没有采用遗传算法,而是利用了 4 条比较简单的测试用例生成规则来生成测试用例.例如,如果将外部输入的字符数组 src 拷贝给缓冲区 dst,那么测试用例生成时要分别生成 src 长度为 *size*,*size*+1,2×*size* 的字符串,其中,*size* 是缓冲区 dst 的大小.该方法是轻量级的缓冲区溢出检测方法,比 Haller 等人的方法时空开销小很多.但是该方法也无法保证路径覆盖率.

- 静态分析协同检测的缓冲区溢出漏洞动态测试技术

Babic 等人[36]采用先动态分析,再静态检测,之后又动态检测的方法.该方法首先运行测试用例将间接跳转标记出来,再执行跨函数的、上下文敏感的、流敏感的静态分析,找出候选缓冲区溢出缺陷,最后再执行动态检测,对静态检测的结果进一步筛选[36].该方法的优点在于静态检测的结果较为精准,使得动态检测的漏报率较低,弥补了常规动态测试的不足.不足之处在于其对大型程序的检测吞吐性较弱.

Bindu 等人[37]采用的是动态测试和静态检测结合的分析手段,该方法首先通过静态分析技术找到所有可能发生缓冲区溢出的槽节点,然后针对这些槽节点构造测试用例.若动态测试证明某槽节点确实有可能发生缓冲区溢出漏洞,则直接将该节点归为溢出节点;若动态测试无法证明某槽节点是否可能发生缓冲区溢出漏洞,则再利用静态检测的方法对该节点进行分析判断.该方法的好处是在保障了误报率较低的基础上,通过动态测试与静态检测相结合的方法,与纯静态检测方法相比,极大程度地节约了静态检测时的时空开销;与纯动态测试方法相比,提高了测试覆盖率,减少了漏报.

表 3 是上述动态测试方法的对比.

**Table 3** Comparison of dynamic testing approaches

**表 3** 动态测试方法对比

| 方法分类 | | 不利用静态分析 | 静态分析辅助输入 | | 静态分析协同检测 | |
|---|---|---|---|---|---|---|
| 典型代表 | | Wang 等人 (2011) | Haller 等人 (2013) | Bindu 等人 (2015) | Babic 等人 (2011) | Bindu 等人 (2014) |
| 技术特点 | 分析对象 | 拷贝字符串型缓冲区溢出 | 数组下标越界型缓冲区溢出 | 数组下标越界型缓冲区溢出、拷贝字符串型缓冲区溢出 | 数组下标越界型缓冲区溢出、拷贝字符串型缓冲区溢出 | 数组下标越界型缓冲区溢出、拷贝字符串型缓冲区溢出 |
| | 使用技术 | 2008 年提出的 ACTS 技术 | 污染数据传播分析、数据流分析、控制流分析和符号执行等 | 数据流分析、控制流分析 | 符号执行数据流分析 | 符号执行机器学习 |
| 测试能力 | 时空开销 | 中 | 中 | 较低 | 较高 | 中 |
| | 全路径覆盖 | 否 | 否 | 否 | 否 | 否 |
| 主要优点 | | 对于字符串拷贝型的缓冲区溢出漏洞检测漏报率较低 | 对于循环中的数组下标越界导致的缓冲区溢出检测漏报率较低 | 1. 运行开销小<br>2. 数组下标越界型缓冲区溢出和拷贝字符串型缓冲区溢出漏洞检测漏报率均较低 | 静态检测的结果较为精准,使得动态检测的漏报率较低,弥补了常规动态测试的不足 | 与纯静态检测方法相比,动态测试帮助减少了人工确认;与纯动态测试方法比,提高了测试覆盖率,减少了漏报 |
| 主要缺点 | | 没有检测数组下标越界型缓冲区溢出 | 没有检测拷贝字符串型缓冲区溢出 | 路径覆盖率较低 | 对大型程序的检测吞吐性较弱 | 仍存在误报与漏报 |

### 2.2 漏洞自动修复技术

软件漏洞的自动修复技术是近年来新崛起的研究领域,不但能发现程序中的软件漏洞,还能自动地修复,使软件能够更安全\可靠地运行.自动修复技术的修复发生时间分为软件发布前和软件发布后.漏洞修复技术可按照修复策略的复杂程度分为 3 种.

1) 采用简单修复策略的缓冲区溢出漏洞修复技术:在进行溢出漏洞修复时,只采取 1 条简单常规修复策略的漏洞自动修复技术.例如,对所有要修复的对象都在访问缓冲区前加入条件判断语句,以确保访问位置在缓冲区分配大小内.
2) 采用复合修复策略的缓冲区溢出漏洞修复技术:在进行溢出漏洞修复时,采取多条常规修复策略的漏洞自动修复技术.例如,在修复过程中,依据不同情况采用扩充缓冲区分配大小、访问缓冲区前增加条件判断、替换可能会发生缓冲区溢出漏洞的不安全函数操作等不同策略进行修复.
3) 采用其他修复策略的缓冲区溢出漏洞修复技术:不像情形 1)中或者情形 2)中使用常规的修复策略,而是采用对软件进行遗传变异或者其他非常规的方法对溢出漏洞进行自动修复.

- 采用简单修复策略的缓冲区溢出漏洞修复技术

Stelios 等人(PLDI 15)[38]采用动态插桩和数据结构转换技术,借鉴生物学的基因移植技术,通过移植捐助者程序的关键基因进行自动修复.该方法的思想是:一个程序具有某个特定性质,说明其具有某种特定基因.那么找到能够避免发生溢出漏洞的捐助者程序中的特定基因(某个条件判断逻辑),再将其移植到待修复的程序中,则待修复的程序就具备了该基因,进而不会发生缓冲区溢出漏洞.该方法的优点在于不需要待修复程序源代码信息和符号信息[38].不足之处在于依赖捐助程序基因库是否足够丰富和优秀、具备待修复程序需要的基因,因而该方法无法保障修复的完备性和正确性.

Stelios 等人(CSAILTR 15)[39]用动态插桩和变量依赖分析技术进行漏洞修复:首先插桩执行记录变量依赖关系,然后依据变量依赖关系生成测试用例并用以触发漏洞,接下来加入条件判断对缓冲区访问索引进行限制,最后利用测试用例验证修复正确性.该方法采用简单修复策略,对缓冲区访问索引进行限制,在修复过程不会影响程序其他部分的逻辑.不足之处是修复对象有局限,仍存在漏修的漏洞.

- 采用复合修复策略的缓冲区溢出漏洞修复技术

Alex 等人[40]提出的方法是软件发布前自动修复方法.该方法首先使用到达定义分析及别名分析检测到由不安全函数引起的疑似缓冲区溢出缺陷,然后采用两条修复策略:或者将函数替换为安全的函数操作,或者将涉及的缓冲区修改为新的安全数据结构(包含缓冲区起始地址、现在所指向的地址、缓冲区长度和现在剩余的长度),从而达到将所有的不安全函数操作替换为安全函数操作的目的[40].该方法修复效率较高.不足之处是安全数据结构替换过程中将数据内存分配空间的大小提高,使得修复后的程序在运行时内存开销较大.

Gao 等人[41]提出的方法是软件发布前自动修复方法.该方法对静态分析工具(该文中是 Fortify)汇报出的缓冲区溢出漏洞警告进一步判断,将其中的误报排除,并对正确警告进行自动修复.该方法的分析基于控制流图,通过对静态分析工具报出来的缓冲区溢出警告发生节点进行逆向分析、符号执行分析,生成路径可达条件和漏洞触发条件取交后的约束,再进行约束求解,将约束可满足的路径归为正确警告,并对其进行自动修复.该方法修复的手段有 3 种:1) 加入对缓冲区边界的条件判断;2) 将不安全的函数替换成安全函数;3) 增大缓冲区内存大小[41].该方法的优点是可作为静态检测工具的补充,高效地判断静态检测工具汇报的信息是否为误报;并且对确定的漏洞提供 3 种方法修复,更贴近程序员人工修复策略[41].不足在于依赖静态分析结果,若静态分析结果漏报率高,则无法高覆盖率地修复程序中的缓冲区溢出漏洞.

- 采用其他修复策略的缓冲区溢出漏洞修复技术

Arcuri[42]提出的方法是软件发布前自动修复的方法.该方法先用测试用例集测试待发布的软件,若测试用例未全通过,则用遗传算法对软件进行变异直到测试用例全部通过为止[42].该方法将生物学中的遗传变异思想应用于计算机学科的缓冲区溢出漏洞自动修复领域,具有一定的创新性和启发性.其不足之处是:即使测试用例全部通过,也无法保证软件中没有软件漏洞,测试用例全部通过并不代表缓冲区溢出漏洞全部被修复成功;且修复过程有可能改变程序其他部分的逻辑.

Chen 等人[43]提出的方法是软件发布后自动修复的方法.该方法优点是在发现缓冲区漏洞后,并未像常见的技术停止程序运行去修复漏洞,而是把有可能发生溢出的栈空间通过虚拟化技术转移到 SafeStack 空间进行运行、补丁生成与修复,保障了软件的正常运行[43].该方法的不足在于只对栈空间进行了保护.

表 4 是上述自动修复方法的对比.

**Table 4** Comparison of automation of fixing approaches

**表 4** 自动修复方法对比

| 方法分类 | | 简单修复策略 | | 复合修复策略 | | 其他修复策略 | |
|---|---|---|---|---|---|---|---|
| 典型代表 | | Stelios 等人 (PLDI 15) | Stelios 等人 (CSAILTR 15) | Alex 等人 (2014) | Gao 等人 (2016) | Arcuir 等人 (2008) | Chen 等人 (2013) |
| 技术特点 | 修复时间 | 软件发布前 | 软件发布前 | 软件发布前 | 软件发布前 | 软件发布前 | 软件发布后 |
| | 使用技术 | 动态插桩 数据结构转换 | 动态插桩、变量控制依赖分析 | 别名分析、到达定义分析 | 静态代码分析、动态符号执行 | 遗传算法 | 内存虚拟化技术 |
| | 保护对象 | 堆、栈、数据段 | 堆、栈、数据段 | 堆、栈、数据段 | 堆、栈、数据段 | 堆、栈、数据段 | 栈 |
| 分析准确度 | 使用测试用例验证 | 是 | 是 | 否 | 否 | 是 | 是 |
| | 保护对象全覆盖 | 否 | 否 | 否 | 否 | 否 | 是 |
| | 使用函数摘要 | 否 | 否 | 是 | 是 | 否 | 否 |
| 主要优点 | | 借助捐助者程序进行自动修复,不需要源码信息和符号信息[38] | 修复一定不会影响程序其他部分的逻辑 | 可以保证几乎消除掉所有不安全函数带来的缓冲区溢出漏洞 | 可作为静态检测工具的补充,高效地判断汇报的信息是否是误报;对确定的漏洞提供 3 种方法修复,更贴近程序员人工修复策略[41] | 采用遗传算法可自动化地反复修复,直到测试用例全部通过[42] | 可以保证在代码运行的同时自动修复代码,对栈缓冲区溢出修复效果较好 |

**Table 4** Comparison of automation of fixing approaches (Contiuned)

**表 4** 自动修复方法对比(续)

| 方法分类 | 简单修复策略 | | 复合修复策略 | | 其他修复策略 | |
|---|---|---|---|---|---|---|
| 典型代表 | Stelios 等人(PLDI 15) | Stelios 等人(CSAILTR 15) | Alex 等人(2014) | Gao 等人(2016) | Arcuir 等人(2008) | Chen 等人(2013) |
| 主要缺点 | 1. 依赖于捐助程序基因是否足够优秀<br>2. 无法保障修复的完备性和正确性 | 修复对象有局限,仍存在漏修的漏洞 | 修复后的程序运行时内存开销增大 | 依赖静态分析结果,若静态分析结果漏报率高,则无法高覆盖率地修复程序的漏洞 | 1. 需要输入形式化规约<br>2. 修复后无法保证没有漏洞<br>3. 有可能改变程序其他部分逻辑 | 只修复了分配在栈缓冲区溢出漏洞 |

### 2.3 漏洞运行时防护技术

第 1.3 节描述了对缓冲区溢出漏洞可能的攻击.由于攻击方式多样,导致的后果严重,漏洞运行时防护技术面向包含漏洞的软件,将其保护起来,从而屏蔽了外部多变的攻击方式.该技术相当于把可能存在缓冲区溢出漏洞的软件放在一个安全的保护罩里去运行.如果保护罩足够安全,那么,即使外部有攻击,也不会攻击成功,而是终止程序或者根本不影响程序的运行.

本文依据运行时防护策略与信息安全三要素(机密性、完整性和可用性)的保障关系,将缓冲区漏洞运行防护技术分为两大类.

1) 被动防护:旨在保障内存中某些关键对象或性质完整性的防护技术.在程序运行过程中,时刻验证监控某些性质是否得到满足或者某些状态值(如栈中的返回地址或者 EBP)是否没被修改.因为时刻在验证关键对象或性质的完整性,当攻击出现时,该技术可以被动地检测到有人尝试在修改某些关键对象或者性质,并在攻击的触发下做相应的防护处理,因此称其为被动防护.

2) 主动防护:旨在保障内存中某些关键对象/性质的机密性/可用性的防护技术.攻击者要想实现攻击,必须对程序运行时的内存某些关键部分(如 EBP 的内存具体位置)布局情况了解,而如果通过某些技术手段保障了这些关键部分的机密性,让攻击者无法顺利地得到其内存位置信息,那么攻击者就很难攻击成功.另一方面,本文的可用性是指可以更安全、可靠地使用,例如更换动态链接库,把所有的函数操作换成更安全的函数操作.当可用性得到保障时,攻击者想要利用缓冲区溢出漏洞进行攻击就变得更加困难.该技术不是在攻击的触发下做相应的防护处理,而是主动地保障关键对象/性质的机密性/可用性,在程序运行时已经对内存布局进行修改或者对函数使用做了替换,提高了攻击者预测具体内存信息的难度(保障机密性)或者无法利用不安全的函数(保障可用性).

#### 2.3.1 被动防护技术

被动防护技术的典型代表有插入 canary 值、存储 RETADDR 值、指针前后加 guardzone 和低脂指针.

- 插入 canary 值

早期经典的运行时防护缓冲区溢出漏洞的技术是 Cowan 等人[44]于 1998 年提出的 StackGuard.该方法在函数调用栈的 RETADDR 和 LOCVAR 之间加一个 canary 值.每次要跳转、执行到 RETADDR 对应地址的指令之前,要先验证 canary 值是否改变:若 canary 值没有改变,则程序正常运行;若 canary 值发生了改变,则中断程序运行.为了防止攻击者伪造 canary 值进行攻击,canary 值的产生方式演化成 3 类:(1) 随机产生 canary 值,以提高攻击者通过伪造 canary 值而通过完整性验证的难度;(2) canary 值中含有字符串终止符,使得攻击者拷贝攻击载荷时无法顺利完成;(3) 计算 canary 值和 RETADDR 的异或值,并将该异或值保存到一个全局表中,在每次使用 RETADDR 值前,都要计算一遍当前 RETADDR 的值和 canary 的异或值,判断其与全局表中对应的值是否一致,以提高攻击者构造的攻击向量通过完整性验证的难度.该方法的局限性是只对栈上的 RETADDR 进行保护.但该方法极大地启发了后人,是缓冲区溢出检测领域里的一个里程碑.

Propolice[45]技术借鉴了 StackGuard 方法.该技术与 Stackguard 不同之处主要有两点:(1) Propolice 重排了局部变量的位置,将局部变量中的数组变量排在最高地址,这样就保障了局部变量不会被缓冲区溢出影响;(2) 没

有把 canary 值放在 EBP 和 RETADDR 之间,而是把 canary 值放在 LOCADDR 和 EBP 之间.这样不仅保护了 RETADDR 的值,还保护了 EBP 的值.该技术作为编译器拓展在 GCC 4.1 版本就已得到了应用.该方法的优点在于能够防御大多数需要修改函数调用栈的溢出攻击.不足之处是:1) 没有防御破坏堆的攻击方式;2) 没有防御更改函数指针的攻击方式;3) 没有防御针对 setjmp/longjmp 的攻击方式;4) 运行开销较大;5) 没有从根本上解决问题.

- 存储 RETADDR 值

后来的 StackShield[46]技术也是针对 RETADDR 进行保护,该技术的保护方法是每次产生 RETADDR 时,就把该值存储到一个全局的表中.每次要跳转执行前,都比较一下栈上的 RETADDR 值和全局表中的值:若相同,则程序正常执行;若不同,则中断程序执行.StackShield 除了对 RETADDR 进行保护以外,还对函数指针进行保护,它采用的策略是,每次函数指针进行解引用之前,都要判断该指针是否指向代码区:若指向代码区,则程序正常执行;若指向非代码区,则中断程序执行[46].该方法有如下优点:(1) 能够防御大多数需要修改 RETADDR 的溢出攻击;(2) 也能较好地防御更改函数指针的攻击.不足之处在于:(1) 没有防御破坏堆的攻击方式;(2) 没有防御修改 EBP 的攻击;(3) 没有防御针对 setjmp/longjmp 的攻击方式;(4) 运行开销较大;(5) 没有从根本上解决问题.

- 指针前后加 guardzone

运行时防护中,ASLR 没有对任何的指针操作进行验证,有较低的运行开销,但同时,ASLR 没有检测指针还是给了攻击者一些机会[47].Hasabnis 等人[47]针对这一问题,主要针对更改类函数指针这类攻击提出了防护方法 LBC.LBC 首先在每个对象前后都插入一个值 guardzone.然后,用静态分析的方法选出发生算术加减的指针.接下来,每当对选出来的指针进行解引用时都进行检查,判断指针解引用对应的值是否等于 guardzone:如果不等,则程序继续运行;如果相等,则提示出现了访问错误.该方法能够防御多数更改类函数指针这类攻击,并且由于静态分析的辅助筛选,只对疑似指针进行插桩,因此软件运行时开销较低.但该方法忽略了访问对象值与 guardzone 值相等的情况.

- 低脂指针

Duck 等人[48]在堆内存上使用低脂指针思想:分配内存时,把缓冲区基地址信息、缓冲区大小信息等元信息映射到分配的指针上,从而高效地实现了对缓冲区的边界检测,防止运行时缓冲区溢出的发生.低脂指针本身是常规的机器指针,与肥指针相比没有更改缓冲区的结构,具有更好的性能和二进制兼容性[48].

随后,Duck 等人[49]扩展了针对堆内存的低脂指针思想到栈内存.由于堆内存分配的限制较少,在堆上可以根据缓冲区的大小对堆区域分区,以实现低脂指针思想.而栈上内存的分配限制较多,他们针对这一问题,综合采用指针镜像、内存别名等技术,在栈内存上也实现了低脂指针,从而可以高效地对堆、栈内存进行缓冲区溢出的检测[49].实验结果表明:该技术增加了 3%的内存开销,在只防御写溢出时有 17%的时间开销,在写溢出、读溢出都防御时有 54%的时间开销[49].该技术对堆栈内存空间的保护与其他技术相比较为完善,对内存改变较小,内存开销较小.但缓冲区元信息在传递过程中由于指针大小的不确定性可能会有差错,导致漏报.另外,该技术没有对数据段上的内存进行保护.

2.3.2 主动防护技术

典型的主动防护技术有更换动态链接库、加密指针型数据、随机化内存地址、去堆栈布局可预测性.

- 更换动态链接库

Baratloo 等人[50]提出了另一种防护缓冲区溢出漏洞的思路 LibSafe:更换函数动态链接库.对于含有容易被攻击者所利用的类似 strcpy,sprintf 等易发生缓冲区溢出的函数的动态函数链接库进行替换,将其替换成相对应的提供缓冲区边界检测的安全的动态函数链接库.该方法只保障了调用已替换的 C 库函数时,不会因为缓冲区溢出改变 RETADDR 和函数指针的值.但除了调用这些库函数以外,还是有很多溢出手段可以改变 RETADDR 的值和函数指针的值.

- 加密指针型数据

Cowan 等人[51]针对其提出的 StackGuard 只对栈空间进行保护的缺点,提出了一套保护指针不被缓冲区溢出攻击利用的方法 PointGuard.该方法的基本思想是:攻击者利用缓冲区溢出漏洞只会溢出修改堆、栈、数据段上的数据,而不会修改存储在寄存器中的数据,内存中如果存放的是加密后的数据,攻击者就无法随心所欲地进行攻击;PointGuard 对指针型数据进行加密后再将其存放在内存中,指针解引用前再在寄存器中解密,这样即使攻击者通过利用缓冲区溢出漏洞修改了内存中的指针数据,解密后的指针指向的真正内存位置也不会是攻击者预想的位置[51].如果把修改指针值的缓冲区溢出攻击比作子弹,这种防御手段起到的效果就是让子弹打偏.虽然可能也会有溢出发生,但子弹不会打中目标对象,即不会跳转到攻击者期待的攻击代码位置,进而溢出攻击被防护.这种方法的缺点是运行开销较大,并且没有从根本上解决问题.

- 随机化内存地址

Bhatkar 等人[52]针对 Ret2libc 攻击方式,提出了检测方法 ASLR:通过对堆、栈、数据段等内存地址的随机化,增加攻击者预测目的地址的难度,提高了缓冲区溢出攻击的难度,同时降低了利用缓冲区溢出漏洞的病毒的传播速度.与 PointGuard 相比,该方法没有对指针的加解密过程,节约了运行开销.Microsoft Office 2007 全面使用了 ASLR 功能,也印证了该方法性能良好[53].ASLR 对跨栈帧攻击防御效果好,由于加入 padding 的大小受限制(如果 padding 值较大,则内存空间容易被占满,不够使用),其对栈帧内攻击防御效果一般.因此,ASLR 没有防住所有的溢出攻击,只是提高了攻击难度,没有从根本上解决问题.

- 去堆栈布局可预测性

Chen 等人[54]认为栈的缓冲区溢出层出不穷的本质原因是堆栈布局的可预测性,而可预测性的本质原因是栈空间是线性增长的,因此提出了防护技术 StackArmor.传统的 ASLR 技术随机化栈的基地址,使得每个栈帧对象地址难以预测,但对于栈帧内部,利用相对位置关系,仍然可以被攻击者利用;进一步完善后的 ASLR 技术,在栈帧内部缓冲区变量和普通变量也有间隔,但由于内存的局限性,该间隔较小,还是很容易被预测[54].该方法通过对逻辑栈帧先分割再置换的方式,更为彻底地打乱栈帧内变量和变量的分配位置信息、栈帧和栈帧间的位置信息,达到不让逻辑上相邻的栈空间内存在物理上相邻的目的,其栈空间的内存位置预测难度大于 ASLR,对栈空间上跨栈帧/栈帧内攻击防御效果均优于 ASLR.StackArmor 允许用户根据不同的安全需求和性能开销需求去调整防御的强度:其在默认强度下增加了 5%的时间开销,增加 5MB 内存;在全面防护下,增加了 28%的性能开销,增加 112MB 内存[54].该方法的不足在于只对栈上的空间进行保护,且在全面防护下的时空开销较大.

表 5 是以上这 8 种防御技术和被防御对象的内存位置的对应关系,表 6 是上述运行时防护方法的对比.

**Table 5** Correspondence between protection technology and storage location of protected object

**表 5** 防御技术与被防御对象的内存位置对应关系

| 防护技术 | | | 内存位置 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 栈 | | | | 堆 | | 数据段 | |
| | | | 返回地址 | EBP | 局部变量 | | 指针 | 非指针 | 指针 | 非指针 |
| | | | | | 指针 | 非指针 | | | | |
| 被动防护 | 插入 canary 值 | StackGuard | √ | – | – | – | – | – | – | – |
| | | Propolice | √ | √ | √ | √ | – | – | – | – |
| | 存储 RETADDR 值 | StackShield | √ | – | √ | – | √ | – | √ | – |
| | 指针前后加 guardzone | LBC | √ | √ | √ | – | √ | – | √ | – |
| | 低脂指针 | Low-Fat pointers | √ | √ | √ | √ | √ | √ | – | – |
| 主动防护 | 更换动态链接库 | LibSafe | √ | √ | √ | – | √ | – | √ | – |
| | 加密指针型数据 | PointGuard | √ | √ | √ | – | √ | – | √ | – |
| | 随机化内存地址 | ASLR | √ | √ | √ | √ | √ | √ | √ | √ |
| | 去堆栈布局可预测性 | StackArmor | √ | √ | √ | √ | – | – | – | – |

**Table6** Comparison of runtime protection approaches

**表 6** 运行时防护方法对比

| 分类 | 被动防护 | | | | 主动防护 | | | |
|---|---|---|---|---|---|---|---|---|
| 代表 | 插入 canary 值 | 存储 RETADDR 值 | 指针前后加 guardzone | 低脂指针 | 更换动态链接库 | 加密指针型数据 | 随机化内存地址 | 去堆栈布局可预测性 |
| 技术特点 | 通过扩展编译器的方法,在函数调用栈的 RETADDR 附近添加一个 canary 值,确保调用栈的 RETADDR 值不被修改 | 1. RETADDR 值被存储到一个全局的表中,跳转执行前,进行比较.<br>2. 函数指针进行解引用之前,都要判断该指针是否指向代码区[46] | 在内存对象前后都插入 guardzone.当危险指针解引用对应的值等于 guardzone,提示访问错误 | 分配内存时,把缓冲区基地址信息、缓冲区大小信息等元信息映射到分配的指针上,高效地实现了对缓冲区的边界检测[48] | 将含有类似 Strcpy 函数等容易发生缓冲区溢出的动态函数链接库替换成安全的动态函数链接库 | 对指针型数据进行加密后再将其存放在内存中,指针解引用前再在寄存器中解密 | 通过加 padding 的方式随机化堆、栈、数段等内存的位置 | 通过对逻辑栈帧先分割再置换的方式更为彻底地打乱栈帧内变量和变量的分配位置信息、栈帧和栈帧间的位置信息,达到不让逻辑上相邻的栈空间内存在物理上相邻的目的 |
| 主要优点 | 能够防御大多数需要修改函数调用栈的溢出攻击 | 1. 能够防御大多数需要修改 RETADDR 的溢出攻击.<br>2. 也能防御所有更改函数指针的攻击 | 1. 能够防御多数更改类函数指针这类攻击<br>2. 由于静态分析的辅助,运行时开销较低 | 1. 对堆栈内存空间的保护与其他技术相比较为完善<br>2. 对内存改变较小,内存开销较小 | 保障了调用已替换的 C 库函数时,不会发生溢出 | 即使有溢出发生,也不会跳转到攻击者期待的攻击代码位置 | 1. 提高攻击难度<br>2. 运行开销较小 | 对跨栈帧攻击和栈帧内攻击防御效果均较好 |
| 主要缺点 | 1. 没有防御破坏堆的攻击方式<br>2. 没有防御更改函数指针的攻击方式<br>3. 没有防御针对 setjmp/longjmp 的攻击方式<br>4. 运行开销较大<br>5. 没有从根本上解决问题 | 1. 没有防御破坏堆的攻击方式<br>2. 没有防御修改 EBP 的攻击<br>3. 没有防御针对 setjmp/longjmp 的攻击方式<br>4. 运行开销较大<br>5. 没有从根本上解决问题 | 1. 忽略了访问对象值与 Guardzone 值相等的情况<br>2. 没有从根本上解决问题 | 1. 元信息在传递可能会有差错,导致漏报<br>2. 没有对数据段上的内存进行保护<br>3. 没有从根本上解决问题 | 1. 除了调用已替换的库函数以外,仍有很多溢出手段可以改变 RETADDR 的值和函数指针的值<br>2. 运行开销较大<br>3. 没有从根本上解决问题 | 1. 运行开销较大<br>2. 没有从根本上解决问题 | 1. 没有防住所有的溢出攻击,例如对栈帧内溢出防护效果一般<br>2. 没有从根本上解决问题 | 1. 只对栈上的空间进行保护<br>2. 在全面防护下的时空开销较大<br>3. 没有从根本上解决问题 |

## 3 分析讨论

本文所介绍的缓冲区溢出分析技术分类概览如图 2 所示,以下分别对上述技术进行分析讨论.

缓冲区溢出的漏洞触发点比较明确,对应为发生越界解引用的语句.现有的静态检测技术可以将缓冲区溢出越界解引用的位置以及缓冲区定义的位置和变量传递的位置等信息提供给软件的开发者,有利于从根本消除缓冲区溢出漏洞.但主要问题有:

(1) 缓冲区溢出相对于其他类型的漏洞,分析的精确度在很大程度上取决于区间分析的准确度.由于静态检测技术的不可判定性,且无法获得程序运行时信息,因此,现有静态技术只能对访问缓冲区的索引区间进行上近似或下近似,所以难免会产生误报和漏报.

(2) 在保证误报率和漏报率较低的同时,要高效地对缓冲区溢出漏洞进行检测,需要对缓冲区指针以及缓冲区的索引分别建立静态依赖图,追踪值的传递.然而在大型程序中,跨函数的值传递普遍存在,且依赖关系复杂,如何保证吞吐量足够大也是一大技术难点[55].

针对上述问题,本文有如下两个观点:(1) 因为造成严重后果的缓冲区溢出漏洞通常是攻击者通过外部输入来获得系统控制权限的,因此污染传播这种针对外部输入数据的传播路径进行跟踪的技术在静态检测缓冲区溢出漏洞这一领域扮演重要的角色,该技术与符号执行或抽象解释(比如区间分析)紧密结合或许可以更准确地检测缓冲区溢出漏洞;(2) 近年来,符号执行、抽象解释、污染传播等传统静态检测技术发展得较为成熟,或许科研人员可以转换思路,暂时不致力于研究提高传统静态技术本身,而是将新兴技术融入传统静态检测技术,比如将机器学习算法用于特征分类,辅助传统静态检测技术,可能会进一步提高缓冲区溢出漏洞静态检测的精度和效率.

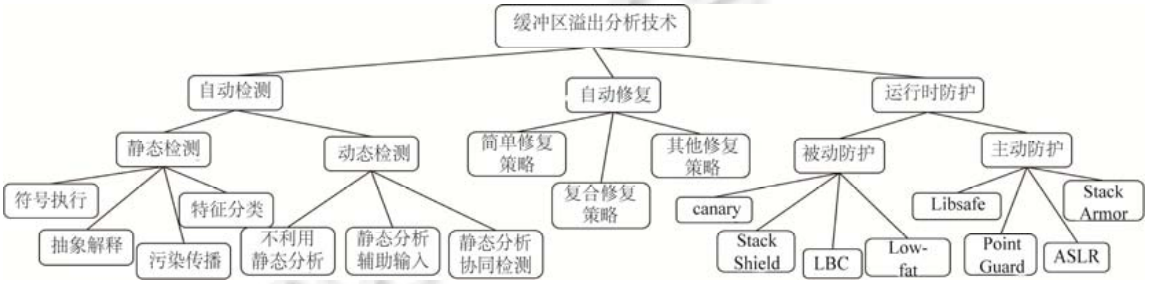



Fig.2 Overview of buffer overflow analysis technologies classification

图 2 缓冲区溢出分析技术分类概览

动态测试基于测试用例生成技术产生测试用例,通过运行程序发现缓冲区溢出漏洞.所发现的缓冲区溢出漏洞一定是真实的,不存在误报.但由于要产生触发缓冲区溢出的测试用例,需要保证访问缓冲区的索引大于缓冲区的大小.为了保证这种值的正确产生和条件的准确判断,动态测试方法在生成测试用例时通常采用符号执行或者遗传算法,因此有生成及运行测试用例时性能开销较大的问题.此外,由于无法做到让测试用例完全覆盖程序中所有可能的缓冲区溢出的程序点,有漏报率较高的缺点.我们认为,如果要更好地提高动态测试的检测能力,应更多地与静态分析结合,即采用静态分析辅助输入或者静态分析协同测试的方法.利用静态分析降低漏报,提高动态测试的覆盖率,利用动态测试去触发缓冲区漏洞,降低误报.例如,将 fuzzing 测试与静态分析结合:Haller 等人利用静态分析得到节点可达性的约束,用其来指导 fuzzing 生成测试用例[34];Mouzarani 等人在 Haller 等人的基础上,把溢出的触发条件与可达约束取交来指导 fuzzing 生成测试用例,比 Haller 等人的方法更 smart,效果更好[56].静态分析能够给出程序本身的部分约束条件,测试能够获得程序的运行信息,动态测试和静态分析恰好能够互补各自的不足.而缓冲区溢出漏洞检测对程序本身的约束信息和运行时信息都需要,动态测试若能恰到好处地借助静态分析,效果就会更理想,即 smarter 甚至优于 smartest.

对软件漏洞的自动修复是近年来新崛起的研究领域,该方法的优点是不但能够发现程序中的软件漏洞,还能自动地修复,使软件能够更安全、可靠地运行.目前,在缓冲区溢出方面的修复有较大问题,使用测试用例验证缓冲区溢出的修复结果.然而测试用例常常不完全,描述的很可能只是缓冲区索引越界的一个子集.因此,能够通过测试用例不代表修复了缓冲区溢出缺陷,无法保障修复的正确性,还可能引入新的缓冲区溢出或逻辑缺陷.软件发布后的自动修复有较大的运行开销,不适合在对实时性要求较高的环境中使用.我们认为,自动修复领域也许会由加入条件判断的单一修复策略方式,逐渐地向采用复合修复策略和采用其他修复策略转化.此外,生物

学中的基因移植[38]、遗传变异[42]等思想也将广泛应用于缓冲区溢出漏洞的修复.随着人工智能技术的崛起,采用其他修复策略的缓冲区溢出漏洞修复技术很有可能会有更好的修复效果.

运行时防护技术能够在一定程度上实时地保障软件不被溢出攻击,具有很高的应用价值.但其只能防御特定模式的溢出攻击,没有从根本上解决问题;另外,检测到溢出攻击时,部分技术采用的应对策略是停止程序运行,虽然可以防止缓冲区溢出的发生,但也较大地影响了软件的可用性;并且,运行时防护技术所增加的运行开销也往往较大.运行时防护技术需要更新颖的方法,尽可能少地修改程序执行环境,以减少正常程序操作期间的开销和不必要的副作用[57].被动防护通常是围绕 EBP、RETADDR、函数指针的完整性或者是缓冲区访问位置需在缓冲区范围内这一性质的完整性展开.目前最新的研究成果低脂指针[49]在对堆内存、栈内存的防护方面已经取得还不错的效果,值得进一步思考与完善的是,如何将该思想扩展到数据段上.主动防护通常是采取主动替换函数链接库、加密函数指针、随机化内存地址、去堆栈布局可预测性等方面,围绕内存的机密性和可用性展开.该技术目前较新的代表是 StackArmor[54],该方法对栈空间位置分配打乱得较为彻底,对栈缓冲区溢出的防护效果较为理想,值得进一步思考和完善的是如何扩充该思想,对堆上及数据段内存的分配规律也进行混淆.此外,被动防护技术与主动防护技术的结合也可能会起到较好的防护效果,但二者结合的内存开销该如何被降到合理范围内,也值得科研人员进一步探讨.

表 7 是上述 4 种分析方法的对比.

**Table 7** Comparison of buffer overflow analysis approaches

**表 7** 缓冲区溢出分析方法对比

| 分析方法 | 漏洞自动检测——静态检测 | 漏洞自动检测——动态测试 | 漏洞自动修复 | 漏洞运行时防护 |
|---|---|---|---|---|
| 技术特点 | 在不运行软件的前提下检测程序中存在的缓冲区溢出漏洞[16] | 从执行域中恰当地选取一组有限的测试用例来运行程序[30] | 不仅能够发现程序中的软件漏洞,还能自动地修复 | 把软件放在一个安全的保护罩里去运行,实时地进行防护 |
| 技术难点 | 1. 如何保证误报率、漏报率均较低<br>2. 如何在保证误报率、漏报率均较低的同时,保证吞吐量足够大 | 1. 如何生成覆盖率高的测试用例<br>2. 如何生成能命中要害、触发缓冲区溢出漏洞发生的测试用例 | 1. 如何降低漏报率<br>2. 如何确保自动修复后,漏洞真正得以排除<br>3. 如何确保自动修复后,程序的原始语义不被修改 | 1. 如何降低运行开销<br>2. 如何防御尽可能多的攻击模式<br>3. 发现攻击后,如何处理以保障程序正常运行 |
| 主要优点 | 1. 在软件发布前发现漏洞<br>2. 可以将漏洞的位置信息汇报给软件的维护者<br>3. 有利于软件开发者从根本上消除缓冲区溢出漏洞<br>4. 代码执行路径覆盖率较高 | 1. 在软件发布前发现漏洞<br>2. 发现的疑似漏洞一定是缓冲区溢出漏洞,无误报 | 1. 发现的疑似漏洞一定是缓冲区溢出漏洞,无误报<br>2. 可以在软件发布前进行修复,降低缓冲区溢出漏洞的存在可能<br>3. 可以在软件发布后进行实时修复,降低运行过程中宕机及被攻击的风险 | 1. 可在软件发布后对其进行实时防护<br>2. 发现的疑似漏洞一定是缓冲区溢出漏洞,无误报 |
| 主要缺点 | 无法获得程序运行时信息,难免会产生误报和漏报 | 1. 生成及运行测试用例时性能开销较大<br>2. 无法做到完全覆盖程序中所有可执行路径,漏报率较高 | 1. 漏报率较高<br>2. 修复后无法保证漏洞真正得到排除<br>3. 无法保证修复后,程序的原始语义不被修改 | 1. 只能防御特定模式的溢出攻击,没有从根本上解决问题<br>2. 检测到溢出攻击时部分方法会停止程序运行,较大地影响了软件的可用性<br>3. 运行开销往往较大 |

## 4 总结与展望

本文首先介绍了缓冲区溢出漏洞危害的严重性和广泛性;然后,从如何利用缓冲区溢出漏洞的角度,依次介

绍了缓冲区溢出漏洞的定义、系统内存组织方式和缓冲区溢出攻击方式;接下来,基于对缓冲区漏洞的理解,根据调研结果将缓冲区溢出分析技术分为 3 类:漏洞自动检测技术、漏洞自动修复技术、漏洞运行时防护技术,并对每一类技术进行了介绍;最后,本文对这 3 类技术进行了分析讨论.

根据调研结果,我们认为,缓冲区溢出分析领域未来有 3 个可能的研究方向.

### 4.1 对二进制代码进行分析

因为很多商业软件不会把源代码暴露给外界,所以对二进制代码的缓冲区溢出漏洞自动检测与修复,很有可能是下一步的研究热点.2016 年 8 月,美国举办的 CGC 挑战赛目标为二进制代码的自动发现、利用及修复,其中一大类漏洞类型为缓冲区溢出[58].自 2009 年以来,程序缺陷修复成为软件工程领域研究的一大热点,涌现出很多不同的自动缺陷修复技术.目前主流技术针对一般类型的缺陷,通过运行测试过滤掉错误的补丁[59,60].针对特定类型的缺陷也有一些研究,如内存泄漏[61]、数据竞争[62]等.但是主要针对源代码进行分析,而对二进制代码漏洞进行检测和修复的研究较少.在缓冲区漏洞这一重要的程序缺陷上,对二进制代码的分析将会更有价值.

对二进制代码的缓冲区溢出漏洞的检测与修复主要有两方面的挑战:第一是对二进制代码进行反汇编,由于很多二进制代码进行了加壳,解析过程有较大困难;第二是保证对缓冲区溢出漏洞修复的正确性,如果一种自动修复方法不能保证修复的正确性,开发与测试人员则不会使用该自动修复方法.同时,必须保证修复补丁不会影响程序的执行效率,因此要尽可能地只在必要的程序位置生成修复.

### 4.2 结合机器学习算法进行分析

随着深度学习算法和人工智能领域近年来的快速发展,科研人员将越来越关注如何结合机器学习和人工智能算法来实现更准确的缓冲区溢出检测.Mou 等人[63]使用深度学习的方法进行研究程序分类问题.Rahul 等人[64]将深度学习应用于程序缺陷修复.我们认为,对缓冲区溢出的检测也将可以使用机器学习技术.

缓冲区溢出有一定的模式,例如对数组、指针的声明、对数组下标的赋值与运算以及对缓冲区溢出位置的数据操作.对缓冲区溢出的检测可以归结为某段代码有无缓冲区溢出漏洞的有无问题,因此转化为机器学习中的二元分类问题.通过使用机器学习算法,可以学习在不同的上下文中出现缓冲区溢出漏洞的模式,进而提高精度.此外,机器学习算法可以用于检测结果的筛选,并和程序分析方法结合使用.例如,首先,通过程序分析方法提供一个可能的缓冲区溢出漏洞列表;再使用机器学习方法精化分析结果,过滤掉检测结果中的误报和漏报.

### 4.3 综合利用多种技术进行分析

如何综合使用漏洞自动检测、漏洞自动修复、漏洞运行时防护这 3 类技术,以获得更准确的检测结果和更低的性能开销,也值得科研人员、工程技术人员进行深入的研究.目前的研究分别在静态检测、动态测试、漏洞自动修复、漏洞运行时防护等方面对缓冲区溢出漏洞进行研究.这些方法各有优劣,可以结合使用,使对缓冲区溢出漏洞检测的精度更高.但是在综合使用的过程中,还可能有各种问题需要解决,例如,如何有效地筛选动态测试中使用的测试用例,使自动修复获得更高的速率;如何使用静态分析的结果帮助运行时防护更有针对性地进行计算,从而节省运行时的防护开销.此外,3 类检测技术若结合使用势必带来更多潜在的效率问题,如何在各个技术的使用阶段整合资源,从而达到最优的并行效果,也是值得研究的问题.

**References**:

[1] The OWASP top ten 2004. https://www.owasp.org/index.php/Top_10_2004

[2] CWE. 2011. http://cwe.mitre.org/

[3] National Computer Network Intrusion Protection Center. Analysis of ten important security breaches in November 2014, 2014 (in Chinese).

[4] National Information Security Vulnerability Library (in Chinese). 2016. http://www.cnnvd.org.cn/

[5] Ye T, Zhang LM, Wang LZ, Li XD. An empirical study on detecting and fixing buffer overflow bugs. In: Proc. of the ICST 2016. 2016. 91−101. [doi: 10.1109/ICST.2016.21]

[6] CVE. 2016. http://cve.mitre.org/

[7] Piromsopa K, Enbody RJ. Buffer-Overflow protection: The theory. In: Proc. of the 2006 Int'l Conf. on Electro/Information Technology. 2006. 454−458. [doi: 10.1109/EIT.2006.252128]

[8] Chen K, Wagner D. Large-Scale analysis of format string vulnerabilities in debian linux. In: Proc. of the Workshop on Programming Languages and Analysis for Security (PLAS 2007). San Diego, 2007. 75−84. [doi: 10.1145/1255329.1255344]

[9] Ahmad K. Classification and prevention techniques of buffer overflow attacks. In: Proc. of the 5th National Conf.; Indiacom-2011 Computing for Nation Development. 2011.

[10] Bishop M, Engle S, Howard D, Whalen S. A taxonomy of buffer overflow characteristics. IEEE Trans. on Dependable and Secure Computing, 2012,9(3):305−317. [doi: 10.1109/TDSC.2012.10]

[11] Wilander J, Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proc. of the NDSS 2003. 2003. 149−162.

[12] Novark G, Berger ED. DieHarder: Securing the heap. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. ACM Press, 2010. 573−584. [doi: 10.1145/1866307.1866371]

[13] Sotirov A. Heap Feng Shui in Javascript. In: Black Hat Europe 2007. 2007.

[14] Padmanabhuni B, Tan HBK. Defending against buffer-overflow vulnerabilities. Computer, 2011,44(11):53−60. [doi: 10.1109/MC.2011.229]

[15] Grieco G, Mounier L, Potet ML, Rawat S. A stack model for symbolic buffer overflow exploitability analysis. In: Proc. of the ICST Workshops 2013. 2013. 216−217. [doi: 10.1109/ICSTW.2013.33]

[16] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697−1708 (in Chinese with English abstract).

[17] Wagner D, Foster J, Brewer E, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In: Proc. of the Network and Distributed System Security Symp. San Diego, 2000. 3−17.

[18] Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities. In: Proc. of the 10th Usenix Security Symp. Usenix, 2001.

[19] Yang ZH, Gong YZ, Xiao Q, Wang YW. The application of interval computation in software testing based on defect pattern. Journal of Computer-aided Design & Computer Graphic, 2008,20(12):1630−1635 (in Chinese with English abstract).

[20] 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]

[21] Wang YW, Yao XH, Gong YZ, Yang ZH. A method of buffer overflow detection based on static code analysis. Journal of Computer Research and Development, 2012,49(4):839−845 (in Chinese with English abstract).

[22] Brat G, Navas JA, Shi N, Venet A. IKOS: A framework for static analysis based on abstract interpretation. In: Proc. of the SEFM 2014. 2014. 271−277. [doi: 10.1007/978-3-319-10431-7_20]

[23] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the CGO 2004. 2004. [doi: 10.1109/CGO.2004.1281665]

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

[25] Le W, Soffa ML. Marple: Detecting faults in path segments using automatically generated analyses. ACM Trans. on Software Engineering and Methodology, 2013,22(3):18:1−18:38. [doi: 10.1145/2491509.2491512]

[26] Ding S, Tan HBK, Liu KP, Chandramohan M, Zhang HY. Detection of buffer overflow vulnerabilities in C/C++ with pattern based limited symbolic evaluation. In: Proc. of the COMPSAC Workshops 2012. 2012. 559−564. [doi: 10.1109/COMPSACW.2012.103]

[27] Gao F, Chen T, Wang Y, Situ L, Wang L, Li X. Carraybound: Static array bounds checking in C programs based on taint analysis. In: Proc. of the 8th Asia-Pacific Symp. on Internetware. ACM Press, 2016. 81−90. [doi: 10.1145/2993717.2993724]

[28] Padmanabhuni BM, Tan HBK. Predicting buffer overflow vulnerabilities through mining light-weight static code attributes. In: Proc. of the ISSRE Workshops 2014. 2014. 317−322. [doi: 10.1109/ISSREW.2014.26]

[29] B Padmanabhuni BM, Tan HBK. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In: Proc. of the COMPSAC 2015. 2015. 450−459. [doi: 10.1109/COMPSAC.2015.78]

[30] Patton P. Software Testing. 2nd ed., SAMS, 2005.

[31] Rawat S, Mounier L. Offset-Aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In: Proc. of the ICST Workshops 2011. 2011. 531−533. [doi: 10.1109/ICSTW.2011.9]

[32] Wang WH, Lei Y, Liu DG, Kung DC, Csallner C, Zhang DZ, Kacker R, Kuhn R. A combinatorial approach to detecting buffer overflow vulnerabilities. In: Proc. of the DSN 2011. 2011. 269−278. [doi: 10.1109/DSN.2011.5958225]

[33] Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J. IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing. Software Testing, Verification and Reliability, 2008,18(3):125−148. [doi: 10.1002/stvr.381]

[34] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. of the USENIX Security 2013. 2013. 49−64.

[35] Padmanabhuni BM, Tan HBK. Light-Weight rule-based test case generation for detecting buffer overflow vulnerabilities. In: Proc. of the AST@ICSE 2015. 2015. 48−52. [doi: 10.1109/AST.2015.17]

[36] Babić D, Martignoni L, McCamant S, Song, D. Statically-Directed dynamic automated test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. ACM Press, 2011. 12−22. [doi: 10.1145/2001420.2001423]

[37] Padmanabhuni BM, Tan HBK. Auditing buffer overflow vulnerabilities using hybrid static-dynamic analysis. In: Proc. of the COMPSAC 2014. 2014. 394−399. [doi: 10.1109/COMPSAC.2014.62]

[38] Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard, M. Automatic error elimination by horizontal code transfer across multiple applications. ACM SIGPLAN Notices, 2015,50(6):43−54. [doi: 10.1145/2813885.2737988]

[39] Sidiroglou-Douskos S, Lahtinen E, Rinard M. Automatic discovery and patching of buffer and integer overflow errors. Technical Report, MIT-CSAIL-TR-2015-018, Boston: Massachusetts Institute of Technology, 2015.

[40] Shaw A, Doggett D, Hafiz M. Automatically fixing C buffer overflows using program transformations. In: Proc. of the DSN 2014. 2014. 124−135. [doi: 10.1109/DSN.2014.25]

[41] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM Press, 2016. 786−791. [doi: 10.1145/2970276.2970282]

[42] Arcuri A. On the automation of fixing software bugs. In: Proc. of the CSE Companion 2008, Vol.I. 2008. 1003−1006. [doi: 10.1145/1370175.1370223]

[43] Chen G, Jin H, Zou DQ, Zhou BB, Liang ZK, Zheng WD, Shi XH. SafeStack: Automatically patching stack-based buffer overflow vulnerabilities. IEEE Trans. on Dependable and Secure Computing, 2013,10(6):368−379. [doi: 10.1109/TDSC.2013.25]

[44] Cowan C. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the USENIX Security 1998. 1998. 63−78.

[45] Etoh H. GCC extension for protecting applications from stack-smashing attacks. 2000. http://www.trl.ibm.com/projects/security/ssp/

[46] Vendicator. Stack shield technical info file v0.7. 2001. http://www.angelfire.com/sk/stackshield/

[47] Hasabnis N, Misra A, Sekar R. Light-Weight bounds checking. In: Proc. of the CGO 2012. 2012. 135−144. [doi: 10.1145/2259016.2259034]

[48] Duck GJ, Yap RHC. Heap bounds protection with low fat pointers. In: Proc. of the 25th Int'l Conf. on Compiler Construction (CC 2016). ACM Press, 2016. 132−142. [doi: 10.1145/2892208.2892212]

[49] Duck GJ, Yap RHC, Cavallaro L. Stack bounds protection with low fat pointers. In: Proc. of the Symp. on Network and Distributed System Security. 2017.

[50] Baratloo A, Singh N, Tsai T. Transparent run-time defense against stack smashing attacks. In: Proc. of the 2000 USENIX Technical Conf. San Diego, 2000. 251−262.

[51] Cowan C, Beattie S, Johansen J, Wagle P. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In: Proc. of the USENIX Security 2003. 2003. 91−104.

[52] Bhatkar S, DuVarney DC, Sekar R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proc. of the USENIX Security 2003. 2003.

[53] Use of ASLR, NX. 2008. http://blogs.msdn.com/david_leblanc/archive/2008/03/14/use-of-aslr-nx-etc.aspx

[54] Chen X, Slowinska A, Andriesse D, Bos H, Giuffrida C. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In: Proc. of the NDSS. 2015. [doi: 10.14722/ndss.2015.23248]

[55] Shahriar H, Zulkernine M. Classification of static analysis-based buffer overflow detectors. In: Proc. of the SSIRI (Companion) 2010. 2010. 94−101. [doi: 10.1109/SSIRI-C.2010.28]

[56] Mouzarani M, Sadeghiyan B, Zolfaghari M. Smart fuzzing method for detecting stack-based buffer overflow in binary codes. IET Software, 2016,10(4):96−107. [doi: 10.1049/iet-sen.2015.0039]

[57] Shahriar H, Zulkernine M. Classification of buffer overflow vulnerability monitors. In: Proc. of the ARES 2010. 2010. 519−524. [doi: 10.1109/ARES.2010.15]

[58] CGC. 2016. https://www.darpa.mil/program/cyber-grand-challenge

[59] Le Goues C, Nguyen TV, Forrest S, Weimer W. Genprog: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2012,38(1):54−72. [doi: 10.1109/TSE.2011.104]

[60] Long F, Rinard M. Staged program repair with condition synthesis. In: Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM Press, 2015. 166−178. [doi: 10.1145/2786805.2786811]

[61] Li X, Zhou Y, Li MC, Chen YJ, Wang LZ, Li XD. Automatically validating static memory leak warnings for C/C++ programs. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):827−844 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5189.htm [doi: 10.13328/j.cnki.jos.005189]

[62] Lu S, Tucek J, Qin F, Zhou YY. AVIO: Detecting atomicity violations via access interleaving invariants. ACM SIGARCH Computer Architecture News, 2006,34(5):37−48. [doi: 10.1145/1168919.1168864]

[63] Mou LL, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence (AAAI). 2016. 1287−1293.

[64] Gupta R, Pal S, Kanade A, Shevade S. DeepFix: Fixing common C language errors by deep learning. In: Proc. of the AAAI 2017. 2017. 1345−1351.

**附中文参考文献**:

[3] 中国科学院大学国家计算机网络入侵防范中心.2014 年 11 月十大重要安全漏洞分析.2014.

[4] 国家信息安全漏洞库. http://www.cnnvd.org.cn/

[16] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1708.

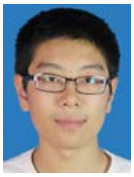
[19] 杨朝红,宫云战,肖庆,王雅文.基于缺陷模式的软件测试中的区间运算应用.计算机辅助设计与图形学学报,2008,20(12):1630−1635.

[20] 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]

[21] 王雅文,姚欣洪,宫云战,杨朝红.一种基于代码静态分析的缓冲区溢出检测算法.计算机研究与发展,2012,49(4):839−845.

[61] 李筱,周严,李孟宸,陈园军,王林章,李宣东.C/C++程序静态内存泄漏警报自动确认方法.软件学报,2017,28(4):827−844. http://www.jos.org.cn/1000-9825/5189.htm [doi: 10.13328/j.cnki.jos.005189]

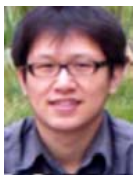
**邵思豪**(1993－),男,辽宁沈阳人,博士生,主要研究领域为信息安全,软件保障.



**马骁**(1994－),男,博士生,主要研究领域为程序分析.

**高庆**(1989－),男,博士,助理研究员,主要研究领域为软件分析,信息安全.

**张世琨**(1969－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件工程,信息安全.



**马森**(1989－),男,博士,助理研究员,主要研究领域为软件分析,信息安全.

**胡津华**(1961－),女,高级工程师,主要研究领域为信息安全.

**段富尧**(1995－),男,博士生,主要研究领域为程序分析.