第 42 卷 第 6A 期
2015 年 6 月

计 算 机 科 学
Computer Science

Vol. 42 No. 6A
June 2015

基于 Libsafe 库的缓冲区溢出检测算法改进

谢汶兵[1,2] 姜 军[1] 李中升[1] 牛夏牧[2]

(江南计算技术研究所 无锡 214083)[1] (哈尔滨工业大学深圳研究生院 深圳 518000)[2]

摘 要 C/C++提供了很多高效的诸如 strcpy 等字符串操作库函数,但由于缺乏相应安全边界检查机制,存在着一些容易被攻击者利用的缓冲区溢出漏洞与威胁。讨论已有的 Libsafe 安全库增强机制并分析了其依赖于栈帧指针回溯栈活动记录的局限性。提出基于程序指令特征码匹配来回溯函数栈活动记录的 Libsafe 安全库增强方法。逐条匹配函数体指令与已知候选集指令来获取栈信息。并在追溯栈活动记录时,将已回溯到的栈活动记录用哈希函数保存,以返回地址作为关键字用链地址法进行检索。对改进版 Libsafe 安全库检测方法做了算法可行性和复杂度分析。从灵敏性、完整性、准确性、性能等几个方面做了实验与分析,表明该机制的高效性与可用性。

关键词 Libsafe 库检测,栈帧指针,堆栈活动记录,指令特征码,回溯栈,哈希函数

中图法分类号 TP309 **文献标识码** A

Improved Algorithm for Buffer Overflow Detection Based on Libsafe Library

XIE Wen-bing[1,2] JIANG Jun[1] LI Zhong-sheng[1] NIU Xia-mu[2]

(Jiangnan Institute of Computing Technology, Wuxi 214083, China)[1]

(Shenzhen Graduate School, Harbin Institute of Technology, Shenzhen 518000, China)[2]

Abstract Due to the lack of boundary checking mechanism of C/C++, buffer overflow is one of the most serious attacks caused by the unsafe functions, such as strcpy. This paper firstly discussed the current mechanism of libsafe library and analyzed the drawbacks using stack frame pointer to look back upon the stack information. We proposed a method through matching the attribute code of instruction's opcode to look back upon the stack information. By matching each opcode with the candidate opcode, we could get the stack information. We also introduced hash function to store the stack information that have been computed and the return address is used as key of the hash function. We analyzed the feasibility and complexity of our improvement algorithm. Experiments were done from different perspectives of cushion, integrity, accuracy. Performance shows the effectiveness of the algorithm.

Keywords Detection of Libsafe library, Stack frame pointer, Activate record, Attribute code of instruction, Look back upon stack, Hash function

1 概述

缓冲区溢出是软件系统中最常见的安全威胁。据 CERT 安全小组统计[1],自 1995 年到 2006 年安全漏洞报告累计达到 30780 个;操作系统中超过 50%的安全漏洞是由缓冲区溢出引起的。2011 年,US-CERT[2]发现了超过 200 个缓冲区溢出漏洞,其中很多是存在于诸如 Microsoft、Adobe 和 Google 等商业软件中。这些与缓冲区溢出相关的安全漏洞正在被越来越多的蠕虫病毒所利用。作为一种有着重要影响的软件安全漏洞,缓冲区溢出漏洞伴随着冯·诺依曼计算机体系结构产生而产生。人们从最初对它的忽视,到后来绞尽脑汁地对它进行研究防范,但是对缓冲区溢出漏洞仍然没有很好的解决办法。发生缓冲区溢出的根本原因是输入到一个缓冲区或者数据保存区域的数据量超过了其容量,从而导致覆盖了其他信息。攻击者造成并利用这种状况使系统崩溃或者插入特制的代码来控制系统。缓冲区溢出攻击的形式有很多种,包括缓冲区破坏攻击、内存指针攻击、帧指针攻击等。

目前,绝大部分操作系统、常用的各种软件都是采用 C/C++语言开发。出于效率考虑,C/C++提供了很多如 strcpy 等字符串操作库函数。而在 C/C++提供灵活的使用方式和高效目标码的同时,这些使用频度很高的库函数在实现时没有加入边界检查代码,缺乏相应的安全机制,存在着一些容易被攻击者利用的安全漏洞,表 1 整理了大量的字符串操作不安全的库函数[3,4]。这些函数的输入来源包括字符串、文件和网络。由于其输入的字符串长度不确定性、长度限制不够和伪长度等特点,如果使用不当,这些函数很容易产生缓冲区溢出问题。

本文受国家"863"高技术研究发展计划项目基金(2012AA010901),国家科技重大专项基(2013ZX01029002),计算机体系结构国家重点实验室开放课题资助。

谢汶兵(1989—),男,硕士生,主要研究方向为安全编译与编译优化,E-mail:xiewb-edu@163.com;姜 军 男,工程师,主要研究方向为编译优化;李中升 男,博士生,高级工程师,主要研究方向为编译优化;牛夏牧 男,教授,博士生导师,主要研究方向为信息安全。

表 1　不安全函数列表

类型	函数
gets()类	gets(),getc(),fgetc(),getchar()
strcpy()类	strcpy(),lstrcpy(),wcscpy(),_tcscpy(),_mbscpy(),strccpy(),strecpy(),strncpy(),lstrncpy(),wcsncpy(),_tcsncpy(),_mbsncpy()
strcat()类	strcat(),strncat(),l strcat(),lstrncat(),wcscat(),wcsncat(),_tcscat(),_tcsncat(),_mbscat(),_mbsncat()
scanf()类	scanf(),sscanf(),fscanf(),vscanf(),vsscanf(),wscanf(),swscanf(),fwscanf(),_tscanf(),vfscanf(),_ftscanf(),_stscanf(),_escanf()
printf()类	printf(),sprint(),fprintf(),vsprintf(),vswprintf(),vsnprintf(),swprintf(),wnsprintf(),_snprintf(),_snwprintf(),_vsnprintf()
其他类	getopt(),getopt_long(),getopt_long_only(),getpass(),getwd(),getpw(),getenv(),OemToChar(),OemToCharBuff() OemToAnsi(),OemToAnsiBuff(),GetTempPath(),syslog(),select(),mbstowcs(),strtrns(),strcadd(),read(),memcpy(),CopyMemory (),bcopy(),strxfrm(),wcsxfrm(),realpath(),chroot(),streadd(),system(),popen()

针对不安全库函数的调用防范,目前提出的一种普遍做法是对不安全库函数进行增强,Libsafe 即为一种最常用的方法。Libsafe 利用动态库的预载机制,封装了若干已知的易受堆栈冲击方法攻击的库函数。Libsafe 通过截获对库函数的调用,监控程序运行时堆栈指针的变化,使用安全的库函数替代有风险的库函数,确保任一缓冲区溢出都被控制在现有堆栈之内[5]。Libsafe 安全库实现可分为 3 个阶段:拦截库函数调用、对库函数调用做安全性检查、漏洞处理。其中对库函数调用做安全性检查为最关键的一步,直接决定对漏洞处理的方式,安全检查是通过回溯栈信息来实现的。本文的改进算法主要是针对对库函数调用做安全性检查这一部分来改进。

本文分析了现有 Libsafe 在对库函数调用做安全性检查机制中依赖栈帧指针回溯栈信息的不足后,提出了基于指令特征码匹配来回溯函数栈信息的改进版 Libsafe 方法。并用哈希函数对该方法进行了优化。为做区分,记已有机制实现的 Libsafe 库为原始版 Libsafe,本文新提出的改进算法实现的 Libsafe 库为改进版 Libsafe。

本文主要贡献概括如下。

(1)详细分析了现有 Libsafe 安全库的实现机制,并对其局限性进行分析。

(2)提出了在对库函数调用做安全性检查时基于指令特征码回溯栈的思想,避免了对栈帧指针的依赖。突破了现有 Libsafe 在做安全性检查时实现思维,使得其有更大的适用范围。

(3)根据(2)提出的思想进行进一步优化,利用程序局部性原理的特点,将已经扫描得到的函数栈信息做了保存。以函数返回地址作为索引值,当有新函数调用时,首先索引已保存栈信息,若检索失败再去进行特征码匹配回溯。

(4)针对(3)中提出的方法,做了进一步优化,引入了哈希函数。用平方取中法进行散列,用链地址法进行索引。使得算法的复杂度大大降低。

本文第 2 节介绍了现有 Libsafe 的实现机制,分别介绍了防范堆栈溢出和格式化字符串的攻击的方法,并分析了现有 Libsafe 实现的不足与局限性;第 3 节详细介绍了改进的 Libsafe 的实现机制,给出了改进算法的方案和实现细节;第 4 节对改进的 Libsafe 算法从可行性和算法复杂度两方面做了分析;第 5 节为实验部分与结果分析;最后为总结。

2　现有 Libsafe 的实现机制

2.1　实现机制

Libsafe 是由著名的 Bell 实验室研制的。其主要用来防范堆栈溢出和格式化字符串的攻击。Libsafe 已经在 Linux 系统上实现并使用。它由一些自行开发的 C/C++语言的库函数组成。当操作系统启动时,通过预载机制把 Libsafe 预先装入。Libsafe 被装入以后,对于被 Libsafe 保护的库函数,就会存在 Libsafe 和标准 libc 库中的两个库函数。当程序执行调用对应的 C/C++库函数时,Libsafe 截获调用请求并根据库函数的类型不同来处理这些请求。一种是调用之前检查存在漏洞的函数(strcpy,strcat 等)是否存在栈溢出攻击;另一种是在被调用函数(scanf、printf 等)执行过程中检查是否存在格式化字符串攻击。然后根据检查结果来处理漏洞,使用安全的库函数替代有风险的库函数或者报警并中断程序的执行。Libsafe 的实现原理基于以下 3 个事实:

1. 溢出一个栈变量,并不必然导致缓冲区溢出攻击。攻击必须在修改程序执行序列的基础上才能实现。

2. 尽管不能防范所有可能的缓冲区溢出攻击,运行时的机制可以防止返回地址被修改,从而可防范缓冲区溢出攻击。

3. 由于指针和局部变量位于堆栈,其地址必须满足一定的条件:局部变量的存取地址范围不能超过栈帧指针的值、不能对大于栈帧指针的内容进行存储操作等[6]。

2.2　安全性检查策略

程序在运行时,内存中会开辟出一块称为栈的连续动态内存区域,用于维护函数调用时所必须的上下文信息。程序可以将上下文信息压入栈,也可以从栈顶弹出。堆栈的大小在运行时由内核动态地调整,堆栈既可以向下增长(向内存低地址)也可以向上增长(向内存高地址),这依赖于具体的实现[7,8]。在讨论中,以 s-machine 架构典型栈结构为例,如图 1 所示。

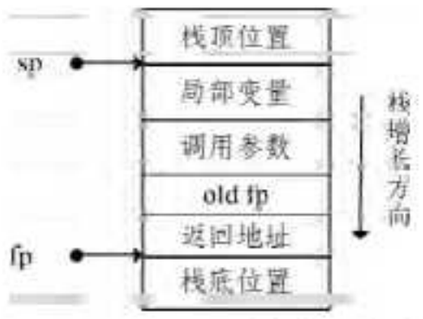

图 1　s-machine 架构典型栈结构

在堆栈中,一个名为堆栈指针(sp)的寄存器指向堆栈的顶部,sp 随着压栈与弹栈的操作而不断变化;栈中还包括:函数的返回地址、调用参数、临时变量和保存的上下文信息等,这些信息统称为栈帧。除了堆栈中的必须信息外,为了便于参数和变量的存取方便,编译器有时还会引入一个固定不变的指针叫做栈帧指针(fp)或者局部基指针(LB-local base pointer)。由于 fp 的位置固定,栈中的局部变量和函数参数到 fp 的距离不会受到栈操作的影响,可以通过 fp 准确定位函数活动记录中的各个参数[9]。

Libsafe 的防范是针对堆栈溢出和格式化字符串攻击进行防范。下面分别介绍两种防范缓冲区攻击的安全检查技术实现[10]。

2.2.1　Libsafe 防范堆栈溢出的安全检查

从如图 1 所示的堆栈中可以看出,栈中依次存放的是函

数返回地址、旧的帧指针、调用参数、局部变量等。函数返回地址、旧的帧指针是固定不变的控制流信息;调用参数、局部变量则为可能改变的数据流信息,其增长方向与栈的增长方向相反。这些数据流信息的增长,可能覆盖上一层函数栈的控制流信息,故需要给数据流信息一个使用上限。可以肯定的是调用参数、局部变量的增长不能超过上一层函数栈底所指的范围,这样就给其限定了一个上限。如进程中调用函数 strcpy(buffer,string),其中定义字符串变量 buffer 所在的函数栈布局如图 2 所示。

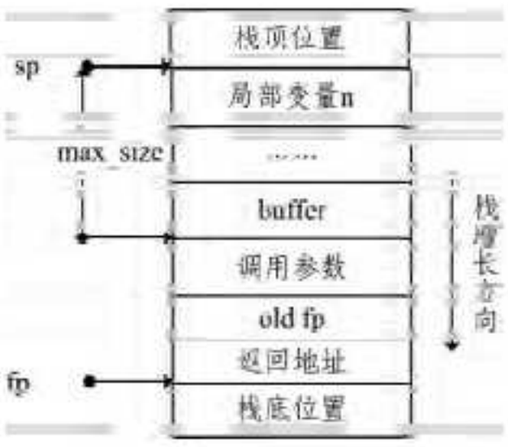

图 2 s-machine 架构函数栈布局

直观上观察其允许的最大拷贝空间为 max-size,也就是说 string 拷贝的字符串长度不得超过 max-size,否则将会覆盖局部变量 n 上一层的返回地址,可能发生缓冲区溢出攻击而导致函数的执行流程错误。而如何找到定义变量 buffer 所在的函数栈信息以及如何知道 buffer 的最大可拷贝空间是一个难题。

目前计算这些信息的方法是通过栈帧指针逐层回溯栈的调用信息来获取的。在栈中越晚被调用的函数离栈顶越近,所以通过栈信息生成函数调用记录时,是从栈顶开始向栈底追溯,这个过程被称为回溯栈[12]。

当一个进程被调用时,所做的第一件事是在栈上分配一定数量字节的临时空间,sp 指向新分配栈的栈底;将旧的 fp 压入栈中,可以保证在函数返回时恢复上一个调用函数的 fp 值,即父函数的 fp;建立新的 fp,然后把 sp 复制到 fp。这样就可以从最底端的栈帧开始,顺着 fp 的指向逐层往回去追溯 fp 从而找到每个函数的栈信息。Libsafe 栈回溯信息的方法正是基于上述思想。在 fp 回溯的过程中,位于栈最底端的栈帧对应于 Libsafe 函数本身。Libsafe 函数的 fp 可通过 gcc 函数 —builtin-frame -pointer(0)找到。Libsafe 防范堆栈溢出的实现流程图如图 3 所示(假设 buffer 的起始地址为 addr)。

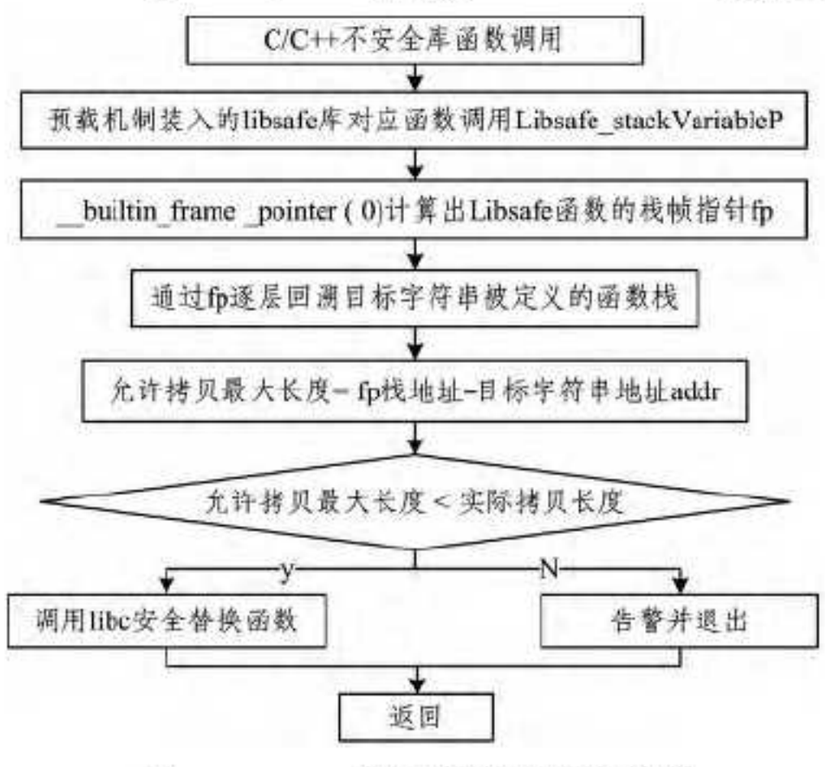

图 3 Libsafe 防范堆栈溢出检测流程

2.2.2 Libsafe 防范格式化字符串的安全检查

*printf()系列函数按照一定的格式对数据进行输出时,可以输出到标准输出,如 printf(),也可以输出到文件句柄、字符串等,如 fprintf、sprintf、snprintf、vprintf、vfprintf、vsprintf 等。而能被黑客利用的地方也就出现在这一系列的 *printf()函数中。正常情况下这些函数不会造成什么问题,但是该系列函数有 3 条特殊的性质:(1)*printf()系列函数参数个数不固定可造成访问越界数据;(2)利用 %n 格式符在提交的字符串当中可放上某个函数的返回地址;(3)可利用附加格式符来控制函数返回地址。这些特殊性质如果被攻击者结合起来利用,就可以组织格式化字符串攻击形成漏洞[11]。

格式化字符串漏洞检测原理针对不同函数的检测方法不同,归纳起来有两种检测类型:1)通过帧指针不断地回溯找到定义目标字符串的函数,然后比较目标字符串可操作的最大空间与源字符串长度,并进行处理;2)处理格式化字符串其他参数,如有 %n 参数,则通过帧指针不断地回溯找到定义目标字符串的函数,并判断目标字符串是否指向函数返回地址或者栈帧指针,再分别作处理。

如—IO—vfprintf()函数,Libsafe 执行以下两个检查操作:①返回地址和栈帧指针检查,对每个 % n,Libsafe 都会检查相关联的指针参数。②栈帧范围检查,确认函数的参数列表位于单个栈帧中。为了执行这两个检查,Libsafe 需确定堆栈中栈帧的位置和大小[12,13]。栈帧信息的获取,和前文提到的防范堆栈缓冲区溢出方法类似。

2.3 现有检查机制局限性

堆栈中,栈帧指针 fp 将一个个的栈帧都串联起来,依靠它可逐级找到前一个函数的栈信息。然而 2.2 节已分析过,fp 的引入是为了更加方便地计算栈中其他变量的位置,除了以下 3 种情况 fp 为必须信息外,其他情况 fp 可以取消掉:a)栈空间比较大;b)有动态申请栈空间的函数调用(如 alloca);c)有自动数组的使用[20,21]。其他情况下有时编译器会取消 fp;且在编译器中,会引入大量的优化选项(如-O2 选项),有时也会考虑将 fp 取消掉。如 gcc 编译器中,-fomit-frame-pointer 选项除了上述 3 种情况外就可以取消 fp,即不使用栈帧指针 fp,计算栈中的其他信息,直接通过 sp 加偏移量来获取[9]。而当编译器优化取消掉 fp 时,尽管栈上还有其返回地址和相关的局部变量数据等信息,但 fp 没有被保存到栈中,无法获取上一个函数的栈信息,给栈的回溯带来了困难。在目前实现的 Libsafe 中,如果在编译程序时加入-fomit-frame-pointer 选项,则即使是一个不安全的函数操作,Libsafe 也无法通过现有机制回溯栈来做检测,而是错误地给出其是安全操作的检测结果。

故现有 Libsafe 基于栈中一个可选项(栈帧指针 fp)来实现栈回溯操作,其受限于编译器中栈帧指针是否存在以及编译器是否优化掉了栈帧指针,有太多的局限性。分析其不足与局限性后,提出了改进版的 Libsafe 库函数调用安全性检查方法,不再依赖于栈帧指针来回溯栈,而是通过分析程序代码段的指令特征来获取栈的上下文信息。

3 改进的 Libsafe 检测算法

3.1 指令特征码分析

在函数调用时,栈的初始化操作一般都需要经过下面几个步骤:(1)栈帧调整:首先保存当前栈帧,将其压入栈,将当前栈帧调整到新栈帧,为新的函数栈帧分配存储空间。(2)返回地址入栈:将当前代码区调用指令的下一条指令存入栈中,函数返回时,可以恢复到调用函数继续执行。(3)参数入栈:

参数按一定的顺序压入堆栈中，调用惯例的约定，函数调用参数从左至右的顺序压入栈。(4)指令代码跳转：指令PC值从当前的函数跳转到相应被调用函数的入口执行。

依据上述函数调用的步骤，函数体开头处对应的汇编指令内容是：

· ldi sp，offset：在栈上分配offset字节的临时空间，sp=sp-offset；

· stl ra，ra-off(sp)：将返回地址保存在(sp+ra-off)的位置；

·【可选】stl fp，fp-off(sp)：将fp保存在(sp+fp-off)的位置；

·【可选】mov sp，fp：fp=sp。

上述代码在每个栈初始化时都是类似的，包括栈空间预留以及栈指针的初始化过程、对FP的赋值。栈空间预留以及栈指针的初始化代码是每个函数体必须的操作，且在每个函数栈初始化过程中，SP最开始时指向该函数的栈底；对FP的赋值则是可选代码。指令及其出现顺序具有固定的模式。

分析程序开头部分栈的初始化过程特点，可以看出如果获取一个调用函数的栈顶指针SP和为该栈分配的临时空间大小，就可以对应知道定义变量所在函数的栈的上下限，从而可知道每个局部变量的最大操作范围。

针对该指令特征码的共性，提取函数体标准开头的指令共同特征码作为候选集保存，实现基于指令特征码的栈回溯的方法。

3.2 回溯调用栈

分析程序代码段特征，每个函数的调用入口都是由上述候选集指令构成，初始化栈的指令除了申请的偏移量offset不同外，其他均是相同的。在s-machine机器上，指令ldisp，offset(sp)的特征码为0xfbdeXXXX(XXXX为偏移量offset的十六进制编码)。类似地，movsp，fp的特征码为0x43fe075e。知道了初始化指令的特征码共性部分，依次分析程序代码段的每条指令特征，在一定条件限制下，将其与上述候选集指令特征码匹配，来回溯每个栈的栈信息内容[14,15]。

例如，针对上面的代码段，根据特征指令序列〈0xadfeXXXX，0x43fe075e〉可判断fp是否存在。若存在则可以直接使用fp，否则需要从指令中获取sp的偏移。根据特征指令序列〈0xfbdeXXXX，0xaf5eXXXX〉来定位修改sp的指令，并通过指令中的偏移量来得到当前函数栈空间的大小。

获取到当前程序指令的pc值，通过它可以知道当前执行指令的具体内容。从该指令开始逐条与候选集里的指令特征码进行匹配。通过候选集的stl fp，fp-off(sp)指令可以知道栈中是否有fp存在，如果存在则可以直接用fp去回溯栈信息。如果没有，则通过匹配候选集stl ra，ra-off(sp)指令可以知道当前函数的返回地址，通过ldi sp，offset可以知道栈帧的栈顶位置以及栈上分配临时空间大小。由函数返回地址可以知道父函数调用该函数的入口地址。通过栈顶指针sp加栈上分配临时空间大小可以知道上一个函数栈的栈底信息。

3.3 Hash算法的引进

依据指令特征码获取栈信息的方法，可以成功回溯到每个调用栈的相关信息。避免了对栈帧指针的依赖，不再受限于函数的栈帧指针是否存在。

但是该方法的缺点也是显而易见的。对于每个无栈帧指针的函数重复调用，都要进行指令的逐条匹配操作。假设函数平均匹配指令序列长度为l，函数平均调用深度为d，函数被调用t次，则总共需要匹配$(l*d*t)$次操作，这是一个可观的性能损耗。如下程序：

```
void f()
{
define dest;
for I=0;i<100;I ++
  g(dest);
}
void g(dest)
{
define src;
strcpy(dest,src);
  ……
}
```

该程序中，函数f()重复调用g()，多次调用的g()函数的栈结构是完全相同的。而在第一次回溯g()调用关系后，就已经获取到了g()的栈帧信息。在接下来的循环中进行g()函数栈信息逐条指令匹配回溯会做大量的冗余工作。

对同一函数的多次调用，其返回地址是唯一的，栈空间结构也是相同的。故考虑将已经回溯的栈信息用链表保存起来，再次调用该函数时，可以直接从链表中取出，避免了不必要的匹配操作。在每次栈回溯的时候，以函数返回地址作为关键字查找链表，如果该函数栈信息已经在链表中，则直接从链表取出对应的栈信息。如果不在链表中，则进行指令匹配，直到找到该栈的信息为止，并将该栈信息加入到新的链表中。

程序调用的局部性原理可知[19]，在程序执行期间，处理器的指令访存和数据访存呈现“簇”状。典型的程序包括许多迭代循环和子程序，一旦程序进入一个循环和子程序执行，就会重复访问一个小范围的指令集合。通过对大量的程序分析，得到函数调用最大深度可以达30次左右，也就是说函数的调用层次一般不是很深，其大多都是局部重复调用。针对函数的调用层次浅、局部调用多的特点，如何进行链表的检索将直接影响Libsafe对函数的整体性能的影响。本文提出用哈希方法来实现。

哈希作为一种重要的存储方法，也是一种重要的查找方法。哈希基本思想是：在记录(元素)在表中的存储位置和该记录的关键字之间建立一种映射关系，关键字的值在这种映射关系下的像，就是相应记录在表中的存储位置。哈希方法需要解决的两个关键问题是：构造哈希函数即确定存储位置；解决冲突的方法与查找操作。

构造哈希函数考虑用平方取中的方法来实现散列。该方法的思想：对于一个关键字k，首先计算其平方值k^2，平方可扩大相近数的差别，使地址值与关键字的每一位都相关，然后根据表长取中间几位(b位)作为散列值，这样可以散列到2^b个位置。

接着需要研究的是如何解决冲突问题。当有不同的关键字被散列到同一个地址集时会有冲突产生，考虑用链地址法解决。具体做法是：将所有关键字为同义词的结点链接在同一个单链表中。将散列表定义为由B(表长$B=2^b$)个单链表头指针组成的指针数组$F[0,1,\cdots,B-1]$。凡是散列地址为

$_i$ 的结点，均插入到 $F[i]$ 为头指针的单链表中。其中 F 个单元的初值均应为空。故改进后的安全增加库检测算法流程如图 $_4$ 所示。

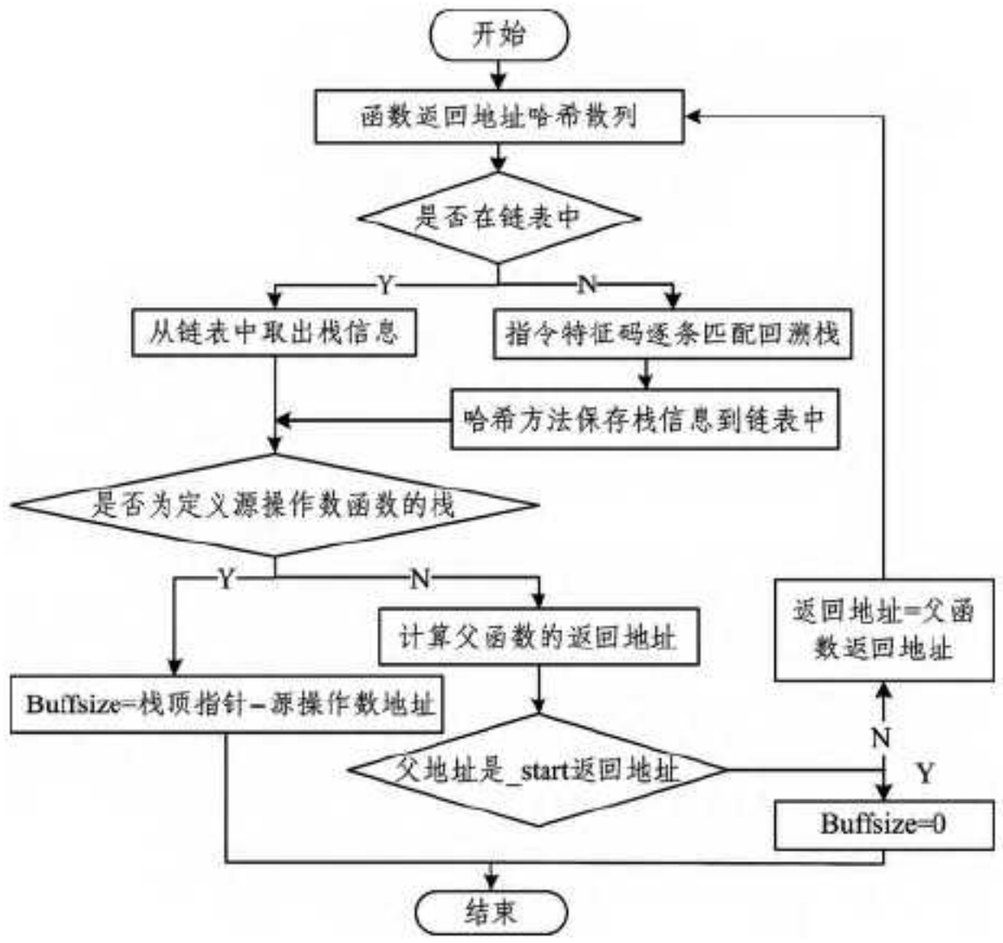

图 $_4$ 改进版 Libsafe 库安全检测算法流程图

4 改进的 Libsafe 检测算法评价

4.1 算法可行性分析

改进的 Libsafe 安全函数检查算法，是从栈最原始的设计出发，依次去回溯栈中的信息。基于指令特征码回溯栈的思想，避免了对栈帧指针的依赖，故该方法对不同的编译器优化都有通用性。

指令特征匹配方法中指令获取比较容易实现。匹配算法也都是成熟的。在实现上原理简单，技术难点比较少。当在新的体系结构上移植时，用户或厂商只要根据该体系结构修改候选集即可使用。

4.2 算法复杂度分析

该算法时间消耗主要是指令特征匹配查找和哈希查找两部分。

指令匹配部分是一个线性查找的过程。其依赖于被匹配函数指令序列的长短与函数调用深度。若每个函数平均指令序列长度为 $_l$，而平均调用深度为 $_d$ 个，则每次调用总共要匹配 $_{l*d}$ 条指令可将链表建立起来，这是一个小常数。

哈希查找部分，若散列的桶长度为 $_B$，则平均每次查找长度为 $_{d/B}$，假如函数总共调用 $_t$ 次，则总的查找次数为 $_{t*d/B}$。

故该算法的总的平均复杂度为

$$l*d+t*d/B=O(t) \tag{1}$$

其中，$_l$ 和 $_d$ 为常数，式 $_{(1)}$ 的复杂度为一个线性增长函数。与 $_{3.3}$ 节提到的总共要匹配 $_{l*d*t}$ 次相比，性能提升在 $_{(l*d*t)/(l*d+t*d/B)\approx l*B}$ 倍左右，复杂度有明显的改善。

5 实验结果与分析

一个好的防御检测工具应该具有较高的准确性和完整性，以及较好的检测效率。即它应当能够在较短时间内找出其中的漏洞。针对上述目标，实验分两步进行，第一步是灵敏性与完整性测试，第二步是准确性与性能测试。

实验环境为 Red Hat 3.4.6，内核版本号为 2.6.28.10，编译器选用 gcc4.8.0，Libsafe 库选择 Libsafe-2.0-16 版本。

5.1 灵敏性与完整性测试

由于目前好多软件漏洞都是未知的，且并不是表 $_1$ 中所罗列的 Libsafe 每个不安全函数都会被覆盖到，故考虑手动编写一些具有针对性的不安全函数调用来验证 Libsafe 的灵敏性和完整性。测试内容包括了堆栈缓冲区溢出攻击、潜在堆栈缓冲区溢出攻击、格式化字符串攻击、潜在格式化字符串攻击几部分（潜在攻击是指攻击覆盖了栈中的数据流信息，但是未覆盖控制流信息）。判断检测结果的依据：对不安全的函数调用，如果有缓冲区溢出攻击发生，其是否会告警并中断；对于有潜在不安全的函数调用时候，其是否会用相应的安全库函数进行替换操作。

此部分总共编写了 $_{35}$ 个测试用例，其中囊括了 $_{26}$ 种不同的不安全函数调用。测试情况具体如表 $_2$ 所列。

表 $_2$ 灵敏性与完整性测试结果

类型	不安全函数调用	测试用例数量	理论检测结果		实际检测结果		
			替换	告警并退出	替换	告警并退出	未检测出
gets()类	gets(),getc()	5	2	3	1	3	1
strcpy()类	strcpy(),lstrcpy(),strccpy(),strecpy(),strncpy(),lstrncpy(),wcsncpy()	10	4	6	3	5	2
strcat()类	strcat(),strncat(),strcat(),wcscat()	3	1	2	1	2	0
scanf()类	scanf(),sscanf(),fscanf()	5	2	3	1	2	2
printf()类	Printf(),sprint(),fprintf(),vsprintf(),vswprintf(),vsnprintf,—vsnprintf()	8	3	5	3	4	1
其他类	getwd(),getpw(),getenv()	4	2	2	1	1	2

分析表 $_2$ 可以看出，其灵敏度和完整性整体较好，$_{35}$ 道测试用例中 $_{27}$ 道被很好地检测出来，被替换为安全函数或者告警并退出。分析改进版 Libsafe 未检测出不安全调用的主要原因是在做库函数处理时，只是针对动态库中封装存在的函数进行操作，对于未封装的函数，则默认依然用 libc 原有的函数，不做任何操作。而 Libsafe 库大小有限，只封装了一定数量的不安全库函数。

此部分同时做了对比试验，用改进版 libsafe 方法与原始版 Libsafe 方法对上述测试用例进行检测。如果编译时加-fo-mit-frame-pointer 选项取消掉栈帧指针，则原始版的 libsafe 除 $_2$ 道题目被检测出外其余全部失败。而改进版 libsafe 检测结果则不受该选项影响。

5.2 准确性与性能测试

为了进一步测试改进的 libsafe 在实际应用中的防御缓冲区溢出能力，考虑用一些实际的应用程序做测试。选择了经常被用来做测试的 Bugbench[16] 软件包作为本次实验测试。Bugbench 是 Lu 等人提供的一个软件包，其中收集了一些已知的存在漏洞的软件，给出了漏洞描述和测试用例。

本实验选择了压缩工具 ncompress、文件名转换工具 polymorph、文件解压包 gzip，文件帮助包 man 4 个包。根据测试用例使用一些特殊输入时，检测结果如表 3 所列。

表 3　Bugbench 测试结果

测试	漏洞描述	测试结果
ncompress	压缩文件名过长，存在溢出可能	Overflow caused by strcpy() and exit
polymorph	转换文件名过长，存在溢出可能	strcpy() replaced by memcpy()
gzip	解压文件名过长，存在溢出可能	strcpy() replaced by memcpy()
man	Man 输入内容特殊，存在溢出可能	Overflow caused by strcpy() and exit

测试结果中 ncompress、polymorph 和 man 符合预期结果。gzip 部分并没有检测出有缓冲区溢出发生，而是做了不安全函数 strcpy() 到 memcpy() 的替换。分析发现 gzip 中 strcpy 函数的目的操作数被定义为全局变量，而全局变量不存放在堆栈中。故在回溯栈时无法正常定位全局变量在堆栈中的位置，只是做了部分 strcpy 替换 memcpy 的操作。

改进版的 libsafe 基于指令匹配与哈希的方法，增加了程序执行时间，会对程序时间性能有一定影响。为测试其时间性能损耗问题，选择了部分 spece2000[17] 的题目来做测试。测试从两方面进行：程序受改进版 Libsafe 保护(libsafe)，程序不受 Libsafe 保护(unprotect)，测试结果如图 5 所示。

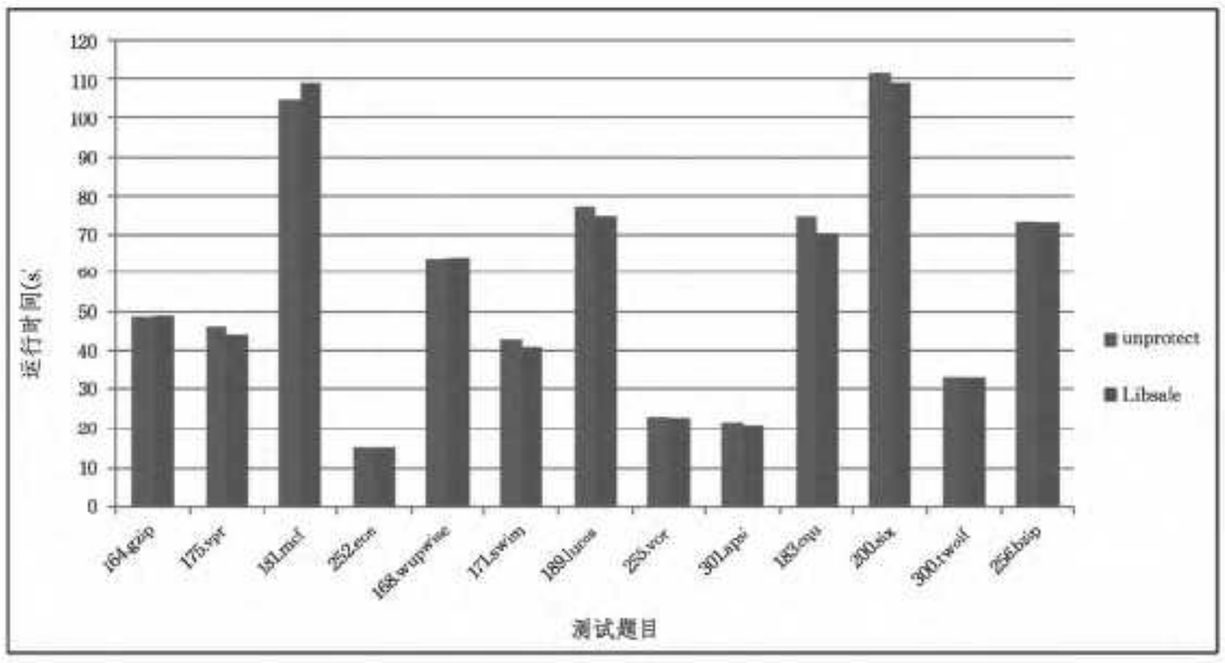

图 5　spec2000 测试结果

在测试 spec 过程中，没有检测到缓冲区溢出的情况，程序全部正常结束。但是遇到不安全的函数调用，其大多都做了相应的函数替换操作。从图 5 可以看出，改进版 Libsafe 在时间性能方面至少与标准库 libc 持平，对一些题目反而有一些提升。分析发现 Libsafe 在进行安全函数检查过程中，如果遇到不安全函数，其会做相应的安全函数替换，如 strcpy 会替换为 memcpy 函数。而对大的缓冲区拷贝而言，memcpy 比 strcpy 要快 6 到 8 倍左右[18]。所以虽然改进版 Libsafe 在动态截获和安全性检查过程中会有一定的性能损耗，但是做了相应函数的替换后，其性能提升可能会更大，故整体上可能会对性能有提升。

结束语　Libsafe 性能高，容易实现，配置简单，只要配置 LD_PRELOAD 环境变量操作系统就可以调用，不会给系统带来额外的负担，不需要重新编译已经存在的应用程序。在性能和兼容性上有相当的优势。通过 strcpy、strcat、realpath、memcpy、printf 等函数族，可以有效防范很多缓冲区溢出漏洞的发生。本文基于指令特征匹配和哈希的方法，避免了对栈帧指针的依赖，可以在更多情况下使用 Libsafe，使其使用范围更广。并且算法复杂度和性能方面也有相当的优势。然而 Libsafe 是基于堆栈检测的，对于全局变量的使用是无法进行检查的，这将是后期研究的一个重点。

参 考 文 献

[1] CNCERT/CC2007 年网络安全工作报告，中国国家互联网应急中心[R]. 2008：11-12

[2] 国家计算机网络入侵防范中心. 2011 年重要安全漏[EB/OL]. http://www.nipc.org.cn/，2012-01

[3] Younan Y，Joosen W，Piessens F. Runtime countermeasures for code injection attacks against C and C++ programs[J]. ACM Computing Surveys(CSUR)，2012，44(3)：17

[4] 何炎祥，吴伟，陈勇，等. 一种用于类 C 语言环境的安全的类型化内存模型[J]. 计算机研究与发展，2012，49(11)，2440-2448

[5] Baratloo A，Singh N，Tsai T K. Transparent Run-Time Defense Against Stack-Smashing Attacks[C]// USENIX Annual Technical Conference，General Track. 2000：251-262

[6] Dhurjati D，Adve V. Backwards-compatible array bounds checking for C with very low overhead[C]// Proceedings of the 28th international conference on Software engineering. ACM，2006：162-171

[7] Vachharajani N，Bridges M J，Chang J，et al. RIFLE：An architectural framework for user-centric information-flow security [C]// 37th International Symposium on Micro architecture(MICRO-37 2004). IEEE，2004：243-254

[8] Shaw A. Program transformatio ns to fix C buffer overflows [C]// Companion Proceedings of the 36th International Conference on Software Engineering. ACM，2014：733-735

[9] 俞甲子，石凡，潘爱民. 程序员的自我修养_链接、装载和库[M]. 北京：电子工业出版社，2009

[10] 潘大庆，覃纪武. 基于 Libsafe 的缓冲区溢出防范技术的研究[J]. 电脑知识与技术：学术交流，2006(7)：86-87

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

[12] 李鹏，王汝传，王绍棣. 格式化字符串攻击检测与防范研究[J]. 南京邮电大学学报：自然科学版，2007，5：1-6

[13] Lin Z，Mao B，Xie L. LibsafeXP：A Practical and Transparent Tool for Run-time Buffer Overflow Preventions[C]// Information Assurance Workshop. IEEE，2006：332-339

[14] 王恩海. 特征匹配引擎设计与实现[J]. 计算机系统应用，2010，19(9)：115-119

(下转第 424 页)

了 AP 的证书，建立了与 AP 之间的基密钥 BK 以及与 ASU 之间的安全通道（K_1 和 $WIE_{asue\text{-}asu}$）。在首次增强型 WAI 证书鉴别过程中，虽然攻击者可以替换 AP 发送给 STA 的 A_{id}，但是并不能形成有效攻击。因为 STA 所接收的 $ECDH_{params}$ 为可识别参数且被用于生成 STA 和 AP 之间的 BK，因此它不可能被替换。此外，虽然 AP 和 ASU 之间的某些参数不为 STA 所知，但是由于 AP 和 ASU 之间的消息不可能被替换，因此这些参数也不可能被替换。

命题 3　假设：①$\sum$为完整的增强型 WAI 证书鉴别过程的串空间，C 为$\sum$中含有一个服务者串 s 的丛，服务者串 s 的迹为 $\mathrm{Serv}[STA, AP, ASU, m_1, m_2, m_5, m_6, m_9]$；② $sk_{asue}, sk_{ae} \notin K_P$；③$x \cdot P, y \cdot P$ 和 $z \cdot P$ 在$\sum$中是唯一产生的。那么 C 中存在一个发起者串 $t \in Init[STA, AP, ASU, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9]$和一个响应者串 $r \in Resp[STA, AP, ASU, m_3, m_4, m_7, m_8]$，其中 $m_8' = flag_2' \| A_{id}' \| ID_{asu} \| Cert_{ae} \| ECDH_{params} \| N_{asu} \| WIE_{asu}$，$m_4' = flag_2' \| A_{id}' \| N_{asue} \| x \cdot P \| ID_{ae} \| Cert_{asue} \| ECDH_{params} \| IDs_{asu} \| WIE_{asue\text{-}asu} \| \sigma_{asue,2} \| \sigma_{asue}'$，$m_7' = flag_2' \| N_{asue} \| N_{ae} \| access' \| x \cdot P \| y \cdot P \| ID_{ae} \| ID_{asue} \| CRes_{cert} \| z \cdot P \| \sigma_{asu,2} \| MAC_{asu\text{-}asue} \| \sigma_{ae,2}'$，$m_8' = flag_2' \| MAC_{asue\text{-}asu} \| MAC_{asue\text{-}ae}'$。

证明：由假设①和②可知，$\sigma_{ae} \subset m_5$ 源发于一个发起者串。根据假设③和定义 1，$y \cdot P \subset \sigma_{ae}$ 唯一产生于这个发起者串。由假设①、②、③和定义 1 可知，$x \cdot P \subset \sigma_{asue,2} \subset m_5$ 唯一产生于一个响应者串。根据假设①和③，$z \cdot P$ 唯一产生于 s。因为定义 1 所指的协议满足沉默性和保守性，因此 $x \cdot y \cdot P$ 和 $y \cdot z \cdot P$ 不源发于 C 中。由于 $K_2 = hash(y \cdot z \cdot P, N_{ae} \| N_{asu} \| str) \notin K_P$，因此 $MAC_{ae\text{-}asu} \subset m_9$ 源发于一个发起者串 $r \in Init[STA, AP, ASU, m_1, m_2, m_3', m_4', m_5, m_6, m_7', m_8', m_9]$。由于 $BK \| A_{seed} = hash(x \cdot y \cdot P, N_{ae} \| N_{asue} \| str)$，从而 $BK \notin K_P$，因此 $MAC_{asue\text{-}ae}' \subset m_8'$ 源发于一个响应者串 $t \in Resp[STA, AP, ASU, m_3', m_4', m_7', m_8']$。由定义 1 可知：$m_3' = flag_2' \| A_{id}' \| ID_{asu} \| Cert_{ae} \| ECDH_{params} \| N_{asu} \| WIE_{asu}$，$m_4' = flag_2' \| A_{id}' \| N_{asue} \| x \cdot P \| ID_{ae} \| Cert_{asue} \| ECDH_{params} \| IDs_{asu} \| WIE_{asue\text{-}asu} \| \sigma_{asue,2} \| \sigma_{asue}'$，$m_7' = flag_2' \| N_{asue} \| N_{ae} \| access' \| x \cdot P \| y \cdot P \| ID_{ae} \| ID_{asue} \| CRes_{cert} \| z \cdot P \| \sigma_{asu,2} \| MAC_{asu\text{-}asue} \| \sigma_{ae,2}'$，$m_8' = flag_2' \| MAC_{asue\text{-}asu} \| MAC_{asue\text{-}ae}'$。

由命题 3 可知：ASU 鉴别了 STA 和 AP，验证了 STA 和 AP 的证书，建立了与 STA 之间的安全通道（K_1 和 $WIE_{asue\text{-}asu}$）以及与 AP 之间的安全通道（K_2 和 $WIE_{ae\text{-}asu}$）。虽然 STA 和 AP 之间的某些参数不为 ASU 所知，但是由于 STA 和 AP 之间的消息不可能被替换，因此这些参数也不可能被替换。

结束语　基于对 TCA 实现的分析，本文指出了现有 WAI 证书鉴别过程不能够很好地支撑 TCA 的平台认证。为了解决这一问题，本文在现有 WAI 证书鉴别过程的基础上提出了一种增强型 WAI 证书鉴别过程，它除实现 WAI 证书鉴别过程的功能外，还可以建立 STA 与 ASU 之间的安全通道，以及 AP 与 ASU 之间的安全通道，而且与现有 WAI 证书鉴别过程是向后兼容的。最后，本文通过串空间模型分析证明了该增强型 WAI 证书鉴别过程是安全的。

参 考 文 献

[1] 黄振海，郭宏，王育民，等. GB15629.11-2003 信息远程通信和信息交换局域网和城域网特定要求第 11 部分：无线局域网媒体访问控制和物理层规范[S]. 北京：中国标准出版社，2003

[2] 宽带无线 IP 工作组. GB15629.11-2003 信息技术系统间远程通信和信息交换局域网和城域网特定要求第 11 部分：无线局域网媒体访问控制和物理层规范和 GB15629.1102-2003 信息技术系统间远程通信和信息交换局域网和城域网特定要求第 11 部分：无线局域网媒体访问控制和物理层规范：2.4GHz 频段较高速物理层扩展规范实施指南[EB/OL]. [2006-01-10]. http://www.chinabwips.org/

[3] 赖晓龙，曹军，铁满霞，等. GB 15629.11-2003/XG1-2006 信息远程通信和信息交换局域网和城域网特定要求第 11 部分：无线局域网媒体访问控制和物理层规范第 1 号修改单[S]. 北京：中国标准出版社，2006

[4] Tang Qiang. On the security of three versions of the WAI protocol in Chinese WLAN implementation plan[C]// Proc of the second International Conference on Communications and Networking in China. Shanghai: ePrint, 2007: 333-339

[5] 铁满霞，李建东，王育民. WAPI 密钥管理协议的 PCL 证明[J]. 电子与信息学报，2009，31(2)：444-447

[6] Trusted Computing Group. TCG trusted network connect architecture for interoperability specification version 1.4[EB/OL]. [2009-05-18]. http://www.trustedcomputinggroup.org/

[7] 沈昌祥，肖跃雷，曹军，等. GB/T 29828-2013 信息安全技术 可信计算规范 可信连接架构[S]. 北京：中国标准出版社，2006

[8] ISO/IEC. ISO/IEC 9798-3:1998/Amd.1:2010 Information technology-Security techniques-Entity authentication-Part 3: Mechanisms using digital signature techniques AMENDMENT 1[S]. ISO/IEC, 2010

[9] Fabrega F J T, Herzog J C, Guttman J D. Strand space: proving security protocols correct[J]. Journal of Computer Security, 1999, 7(2/3): 191-230

[10] Herzog J C. The Diffie-Hellman key-agreement scheme in the strand-space model[C]// Proc of the 16th IEEE Computer Security Foundations Workshop. Pacific Grove: IEEE, 2003: 234-247

(上接第 387 页)

[15] Newsome E J, Karp B, Song D. Polygraph: Automatically generating signatures for polymorphic worms[C]// Proceedings of the IEEE Symposium on Security and Privacy. May 2005: 1-6

[16] Lu S, Li Z, Qin F, et al. Bugbench: Benchmarks for evaluating bug detection tools[C]// Workshop on the Evaluation of Software Defect Detection Tools. 2005: 1-5

[17] Dixit K M. The SPEC benchmarks [J]. Parallel computing, 1991, 17(10): 1195-1209

[18] Avijit K, Gupta P. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection[C]// Proc of 13th USENIX Security Symposium (Security'04). USENIX Association, 2004: 45-56

[19] Denning P J. The working set model for program behavior[J]. Communications of the ACM, 1968, 11(5): 323-333

[20] Han W, Ren M, Tian S, et al. Static Analysis of Format String Vulnerabilities[C]// 2011 First ACIS International Symposium on Software and Network Engineering (SSNE). IEEE, 2011: 122-127

[21] 严芬，袁赋超，等. 防御缓冲区溢出攻击的数据随机化方法[J]. 计算机科学，2011，38(1)：1-5