软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(12):2767−2781 [doi: 10.3724/SP.J.1001.2013.04385]

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

基于信息流的整数漏洞插装和验证*

孙 浩[1,2], 李会朋[1,2], 曾庆凯[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210093)

[2](南京大学 计算机科学与技术系,江苏 南京 210093)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 为降低整数漏洞插装验证的运行开销,提出基于信息流的整数漏洞插装方法.从限定分析对象范围的角度出发,将分析对象约减为污染信息流路径上的所有危险整数操作,以降低静态插装密度.在 GCC 平台上,实现了原型系统 DRIVER(detect and run-time check integer-based vulnerabilities with information flow).实验结果表明,该方法具有精度高、开销低、定位精确等优点.

关键词: 整数漏洞;信息流;污点分析;插装

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 孙浩,李会朋,曾庆凯.基于信息流的整数漏洞插装和验证.软件学报,2013,24(12):2767−2781. http://www.jos.org.cn/1000-9825/4385.htm

英文引用格式: Sun H, Li HP, Zeng QK. Statically detect and run-time check integer-based vulnerabilities with information flow. Ruan Jian Xue Bao/Journal of Software, 2013,24(12):2767−2781 (in Chinese). http://www.jos.org.cn/1000-9825/4385.htm

Statically Detect and Run-Time Check Integer-Based Vulnerabilities with Information Flow

SUN Hao[1,2], LI Hui-Peng[1,2], ZENG Qing-Kai[1,2]

[1](State Key Laboratory for Novel Software Technology(Nanjing University), Nanjing 210093, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

Corresponding author: ZENG Qing-Kai, E-mail: zqk@nju.edu.cn

Abstract: An approach to detecting integer-based vulnerabilities is proposed based on information-flow analysis in order to improve the run-time performance. In this approach, only the unsafe integer operations on tainted information flow paths, which can be controlled by users and involved in sensitive operations, need to be instrumented with run-time check code, so that both the density of static instrumentation and performance overhead are reduced. Based on this approach, a prototype system called DRIVER (detect and run-time check integer-based vulnerabilities with information flow) is implemented as an extension to the GCC compiler and tested on a number of real-world applications. The experimental results show that this approach is effective, scalable, light-weight and capable of locating the root cause.

Key words: integer-based vulnerability; information flow; taint analysis; instrumentation

整型变量是高级语言中常用的基本数据类型,在程序中常用于表示整数数值、内存地址、数组下标、循环计数、标志位或者参与算术运算.由于整数表示形式的局限性和 C 语言类型不安全的特点,整数运算结果失真或由于对整数的不一致解释将导致数据丢失或歧义,这种现象称为整数漏洞.尽管整数错误操作并不会直接对内存做越界修改,但攻击者可通过多种方式间接借助整数漏洞来实现恶意攻击.目前,由整数错误操作导致的脆弱性已遍布于内核代码、实用程序和各种应用中,如 Apache、OpenSSH、Sendmail、Snort、BSD 内核、Linux

* 基金项目: 国家自然科学基金(61170070, 90818022, 61021062); 国家科技支撑计划(2012BAK26B01); 国家高技术研究发展计划(863)(2011AA1A202)

收稿时间: 2012-08-31; 修改时间: 2012-12-03; 定稿时间: 2013-02-04

内核、GNU lib 库函数等[1].据 CVE 统计,整数漏洞一直是 C 语言程序漏洞的主要来源[2].图 1 描述自 1999 年以来 CVE 对整数漏洞的统计结果,近年来整数漏洞急剧增加, 2006 年~2011 年平均数量高达 150 例.按照 RICH[3]的分类方法,整数漏洞可分为上溢(overflow)、下溢(underflow)、符号错误(signedness error)和截断(truncation).图 2 描述 CVE 发布的 2008 年~2011 年 4 类整数漏洞的数量分布,其中,上溢和符号错误最为常见,分别占 77.4%和 13.8%,下溢和截断错误相对较少,二者共占 8.8%.因此,如何高效、精确地检测和处理这 4 类整数漏洞,成为当前的研究热点.

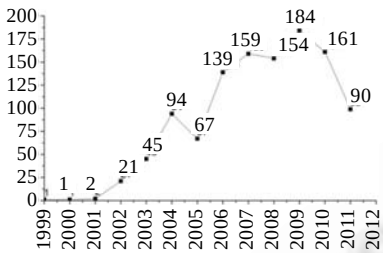


Fig.1 Number of integer-based vulnerabilities reported by CVE from 1999 to 2011

图 1 CVE 对整数漏洞的统计

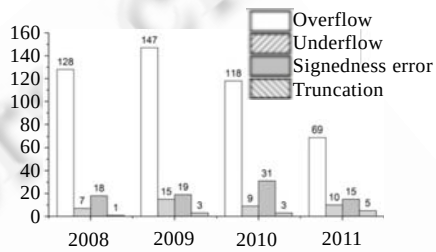


Fig.2 Statistics of integer-based vulnerability categories from 2008 to 2011

图 2 2008 年~2011 年整数漏洞分类统计

在编译器的基础上静态检测出潜在的整数漏洞,并插装验证代码实现动态保护,是针对整数漏洞问题的一种有效解决方案[4].目前,检测插装方法主要有类型规约(如 RICH[3])以及类型推理(如 IntPatch[5]),但存在着运行开销较大、检测精度不高、后续处理不便等不足.本文提出了一种基于信息流来静态检测和动态验证整数漏洞的分析方法,以提高检测精度、降低运行开销.

本文第 1 节介绍相关工作.第 2 节从基本原理、安全模型和安全策略等方面介绍基于信息流的检测方法.第 3 节讨论原型工具的实现细节.第 4 节通过实验对工具的有效性和性能等方面做评估.第 5 节给出总结.

1 相关工作分析

目前,对整数漏洞的研究大致可分为 4 类:

(1) 预防,使用类型安全语言来约束整数操作,如 CCured[6],或者使用安全的库函数来代替整数操作,如 SafeInt[7],AIR model[8],IntegerLib[9]等;

(2) 动态检测,通常采用测试的方法,设计并运行测试用例,触发程序中的错误,如 SmartFuzz[10],Klee[11],Exe[12],Cute[13],Dart[14]等;

(3) 静态检测,通常采用提取程序中整数操作的约束关系来检测可能不满足约束的整数操作,如 Sarkar'07 [15],IntScope[16]等;

(4) 静态插装,由于单纯的静态检测分析会导致大量的误报,在可能发生整数错误的操作点,插装运行时验证代码能有效减少误报,如 RICH[3],IntPatch[5],IOC[4]等.

尽管预防方法能够有效防止整数漏洞,但安全的库函数本身就可能存在整数错误[4],如 SafeInt[7]存在 43 个整数溢出.动态检测方法要产生既能覆盖程序各个分支又能触发整数错误的有效测试用例并非易事,而且由于难以获取精确的类型信息,只能通过有选择地分析可疑指令集来降低工作开销,分析结果不完全.静态检测方法虽能得到较为精确的类型信息,但由于整数错误的触发更多地依赖于用户数据,因而会产生相当高的误报率,影响分析效果.

静态插装方法结合静态检测和动态检测的优点,通过对可能发生错误的整数操作插装验证代码来实现动态保护,一方面有效减少误报,另一方面使运行开销得到缓解,是目前整数漏洞处理的主流方法.如何确定插装位置是静态插装的关键,而漏洞检测精度和性能开销是需要权衡考虑的重要因素.目前,存在类型约束和类型推

理两种主要静态插装方法.

类型约束方法通过形式化描述 C 语言中整数操作的语义规则[3],将类型安全语言中的子类型理论(sub-type theory)[17]应用于 C 语言,对整数的算术运算和类型转换进行整型类型的约束,从而使 C 语言的整数操作具有类型安全的特性.类型规约分析中,首先对所有整数操作进行类型约束求解,对违背类型规约的整数操作插装动态验证代码.例如,RICH[3]和 IOC[4]分别作为 GCC 和 LLVM 的插件得以实现,可针对大量现实软件检测 4 类整数漏洞,并发现多个未知漏洞.该方法虽易于实现,具有较低的漏报率,但是插装密度偏高,导致额外运行开销巨大(其中, RICH 开销为 5%,IOC 开销为 30%).主要原因是类型规约方法选择插装位置的原则过于宽松,一部分整数操作并不会被攻击者利用,或者即使发生错误也不会影响程序的安全性,这部分整数操作不会构成整数漏洞.该方法扩大了插装范围,增加了不必要的静态插装点.

类型推理(type inference)方法沿数据流动方向初始化、传递和转换整型变量的安全等级,最高安全等级的变量被程序安全操作引用便导致脆弱性的发生,所有涉及这部分变量的整数操作被标识为插装位置.与类型规约方法相比,插装位置的确定更为严格,进而使插装密度和运行开销大大降低.例如,IntPatch[5]按照是否可信和是否可能溢出为变量设置 4 种安全类型,随数据流动对变量进行安全类型的传递和转换,当 T_{11} 类型(不可信且可能溢出类型)的变量被内存分配相关函数引用时,认为该引用点是一个 IO2BO 脆弱点;随后,针对每个 IO2BO 脆弱点采用后向切片(backward slicing)技术,逆向沿数据执行流找到所有使用 T_{11} 类型变量的整数操作,并将其判定为潜在的整数溢出漏洞;最后,插装验证代码完成动态保护.IntPatch 作为 LLVM 的插件得以实现,与 RICH 相比有较低的插装密度,额外运行开销仅为 1%.但 IntPatch 只识别可能造成 IO2BO 脆弱性的整数溢出漏洞,不关注其他 3 类整数漏洞.而且,对于由不可信数据通过程序控制流触发的整数漏洞,该方法无法检测.此外,漏洞的定位需要对每个程序安全操作点进行后向切片,静态分析开销较大.

基于以上分析,本文提出一种基于信息流的整数漏洞分析插装方法.信息流包含数据流和控制流,映射程序的语义片段,目前广泛应用于静态或动态检测多种漏洞或提供安全保护[18−23].其中,文献[23]将信息流和污染传播分析结合在一起,提出检测污染型软件脆弱性的通用分析框架,以污染源到程序安全性操作点是否有信息流动作为脆弱性的判定标准,虽能广泛检测多种软件脆弱性(包括缓冲区溢出、内存错误、格式化漏洞、路径遍历等),但由于没有针对整数漏洞的特点来设计,只在程序安全性操作点对污染的整型变量引用时进行检测和保护,难以定位漏洞的根本触发点,对大部分整数漏洞类型无法进行有针对性的精确检测和处理.因此,本文在文献[23]的信息流分析框架的基础上,针对整数漏洞的特性进行扩展和优化,以拓展对整数漏洞的检测能力,降低验证代码的插装密度和运行开销.借助信息流提供的整数类型信息、程序执行流信息和路径信息,本文将类型规约方法和类型推理方法融合在信息流的基础之上,一方面吸取两者的优势,仅对污染信息流路径上的部分安全类型的整数操作进行类型约束求解,可在不影响检测精度的前提下降低插装密度和额外运行开销;另一方面,为弥补两者的不足,不仅将类型规约的分析规模缩减为部分安全类型的整数操作,还可借助信息流提供的控制流信息,将安全类型的传递扩展到控制流上,以检测由不可信数据通过控制流触发的整数漏洞.而且,可通过遍历信息流提供的路径信息方便定位漏洞触发点,避免类型推理中后向切片所需的开销.此外,本文的信息流检测方法针对每个整数漏洞触发点均有污染信息流路径与之匹配,标明用户控制数据的引入点、程序安全性操作点及完整的程序执行路径,有利于程序开发人员对漏洞进行修补.信息流提取对象即用户控制和程序安全性操作点,可按照实际开发需求自由配置,以检测出可能造成各种类型脆弱性的整数漏洞,具有良好的可扩展性.

2 基于信息流的插装和验证

2.1 基本原理

图 3 描述了 CUPS 中存在的整数上溢漏洞 CVE-2008-1722 的程序片段.污染数据由函数 *png_get_IHDR* 引入,经过 *width*/*height*,*img*→*xsize*/*img*→*ysize* 和 *bufsize* 的传递流入内存分配函数 *malloc* 中.当 *width* 或 *height* 足够大时,*bufsize* 处的整数乘法运算可能发生上溢,进而造成内存空间 *in* 远小于预期值,在随后的内存访问过程中可能发生缓冲区溢出错误.函数 *png_get_IHDR* 从外部读取图像文件,将污染数据引入到程序体内,用户可通过

选择不同大小的图像文件作为输入来控制 *width* 或 *height* 的数值,*width* 和 *height* 经过数据流动和算术乘法运算之后,被内存分配函数 *malloc* 引用,进而影响程序的安全性.这里,*bufsize* 的整数乘法运算既能被用户控制又与程序安全性相关,如果发生整数溢出,易于被攻击者利用造成缓冲区溢出的危害.

```
png_get_IHDR(pp,Info,&width,&height,&bit_depth,&color_type,
        &interlace_type,&compression_type, &filter_type);     //untrusted source read from file

img→xsize=width;                                              //propagate
img→ysize=height;

bufsize=img→xsize*img→ysize*3;                                //overflow occurs
in=malloc(bufsize);                                           //used in sensitive operation
```

Fig.3 A real-world integer overflow vulnerability (CVE-2008-1722)[24] in CUPS

图 3 CUPS 存在的整数上溢漏洞 CVE-2008-1722[24]的程序片段

性质 1(危险整数操作). 既能被用户控制又与程序安全性相关的整数操作才有可能成为整数漏洞,这些整数操作统称为危险整数操作.

由于程序中整数操作普遍存在,若对所有的操作语句都进行约束求解和插装,会严重影响时空开销.通过对 CVE[2]公布的 2008 年~2011 年最新的 93 个整数漏洞的研究发现,几乎所有的整数漏洞均具有性质 1 的特点,因此与类型规约方法[3]相比,本文的分析规模可缩减为既能被用户控制又与程序安全性相关的整数操作,即所有的危险整数操作.

信息流是信息从一个实体流向另一个实体的有效途径,前者称为信息流源,后者称为信息流目的地.信息流可以追溯数据的产生、传递路径与使用,有效保存程序的语义信息.本文为信息流实体的污染状态和安全敏感状态定义不同的安全级,若将用户控制数据作为信息污染源,将与程序安全性相关的操作点作为信息目的地,那么可以将用户控制数据在程序中的使用看作安全级在信息流路径上的传递和转换.按照污点分析[22,25,26]的定义,本文的研究对象可规约为以用户控制数据为起点、程序安全性操作点为终点的污染信息流路径上的所有危险整数操作.

值范围传播分析[27,28]可在静态分析阶段得到部分整型变量的可能取值范围,根据危险整数操作中整型变量的可能取值范围和类型信息,可以进一步提高约束求解的精度,从而降低插装数量.因此,本文借助值范围传播分析的结果对污染信息流不同路径上的危险整数操作,按照类型规约[3]对整数操作的详细约束进行求解,排除不可能发生错误的部分危险整数操作.最后,对不满足约束关系的危险整数操作,插装阶段在其之前或之后插入指令级的验证代码以达到动态保护的目的.

综上所述,本文通过两方面降低插装数量,进而降低额外运行开销:

(1) 借助信息流污点分析,提取出所有的危险整数操作作为约束求解的对象;

(2) 借助值范围传播分析,得到危险整数操作中整型变量的可能值范围和类型信息,据此进行静态约束求解,将不满足约束关系的部分危险整数操作作为静态插装的对象.

2.2 信息流安全模型

本文将信息流定义为 *IF*,是信息流识别对象集 *OBJ* 及其上的二元关系→.

定义 1(信息流). $IF=(OBJ,\rightarrow)$,其中,$\rightarrow=\{\langle source,dest,PATHs\rangle|source,dest\in OBJ\}$.

其中,信息流识别对象集 *OBJ* 是一个广义的概念,包括程序中的常数集 *CST*、局部变量集 *LOC*、全局变量集 *GLB*、形参集 *PARM*、函数调用 *FUN* 和返回语句 *RET* 等在内的程序行为实体集合 *OPS*.关系"→"则由信息流源对象 *source* 和目的对象 *dest* 组成,由于程序分支结构、相同实体充当同一操作的不同操作数或函数的不同实参等影响,源和目的对象之间可能存在多条路径,包括直接信息流和间接信息流,记作 *PATHs* 集合,多条信息流构成了信息流集合 *IFs*.

Dening 在程序语义的基础上建立基于格(lattice)的信息流安全模型[29],本文将此扩展到整数漏洞的检测领域,用信息流安全格$\langle L,\cup,\cap\rangle$表示分析对象的安全等级.数据在信息流传递过程中表现为 untainted(清洁)、tainted(污染)、sensitive(敏感)和 vulnerable(脆弱)这 4 种安全级,满足$\perp\preceq untainted\preceq tainted\preceq sensitive\preceq vulnerable$的偏序关系,如图 4(a)所示,其中,$\perp$表示安全级未知,为变量的初始安全级.$L$ 上还定义了程序变量到安全级的映射 $level$:$VARS\rightarrow L$,取数据在当前程序环境下的安全级,其中,变量集 $VARS$ 包括局部变量集 LOC、全局变量集 GLB 和形参集 PRM.

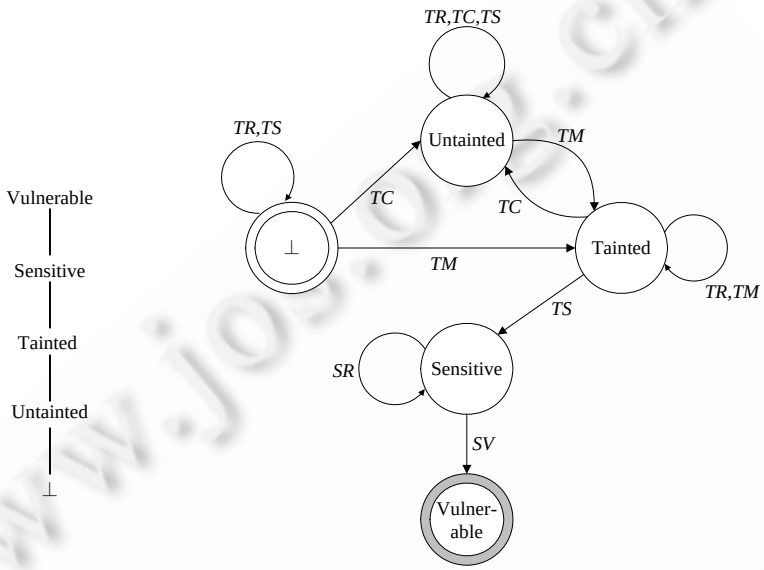


Fig.4 Lattice model and security state transitions in information flow analysis

图 4 基于信息流检测整数漏洞的格模型及安全状态转换

本文将基于信息流检测整数漏洞的安全模型定义为四元组:$M=(IF,L,T,f)$,其中,IF 为程序中的信息流关系;L 为信息流安全格;T 是安全级的传播模式;$f:VARS\times OPS\rightarrow L$ 是一个映射,定义了程序变量及作用于其上的行为实体到变量安全级的转换关系.OBJ 中有 $VARS\subseteq OBJ$ 且 $OPS\subseteq OBJ$,安全级在信息流中通过赋值、运算、函数调用及控制分支的选择等操作,遵循定义 2 的模式 T,在变量间进行传播,而在受到程序行为作用时发生改变.OPS 中存在 6 种程序行为实体,分别对应污染产生、污染清除、污染传递、污染用于敏感操作、敏感传递和敏感违背规约变为脆弱点这 6 个动作,映射 f 中的转换规则见表 1 的 6 种形式.

定义 2(安全级的传播模式). 安全级在信息流 if 中传播,当且仅当$\exists v1,v2\in\mathrm{obj}(if),v1\rightarrow v2$,有 $v1\in OBJ$ 且 $v2\in OBJ$,使得:

$$\begin{cases} level(v2)=level(v1), & level(v2)\preceq tainted\wedge level(v1)\preceq tainted \\ level(v2)=tainted, & level(v2)\preceq tainted\wedge level(v1)=sensitive \text{ or } vulnerable \\ level(v1)=sensitive, & level(v2)=sensitive\wedge level(v1)=tainted \\ \text{do nothing}, & level(v2)=sensitive\wedge level(v1)\neq tainted \\ \text{do nothing}, & level(v2)=vulnerable \end{cases}.$$

$senstive$ 以下的污染状态安全级沿信息流正向传递,$sensitive$ 标识的敏感状态安全级则沿信息流逆向传递,其中,$obj(if)\subseteq OBJ$,取信息流 if 中的识别对象:对于直接信息流,仅返回源和目的对象;间接信息流则按程序执行时的信息流动顺序,从源至目的对象,取出其信息传递路径上的所有实体.

定义 3(整数操作的静态安全约束). 对于整数操作 n,操作数 $m,n\in OPS,m\in OBJ,filter(m,n)$=True 当且仅当对于操作数 m,整数操作 n 满足静态安全约束.

filter 函数用于静态判断某个整数操作是否是安全的,对于静态无法判断的情况,则保守判定为 False.

Table 1 Rules for security state transitions and classification of operation entities

表 1 安全级转换规则及行为实体分类

Rule	Role of entity	State transition	Comments
Taint source rule (R1)	*TM* (taint-maker)	$f(m,n)=tainted,\ m\in VARS\wedge n\in TM$	Functions or operations that import Tainted data are the sources of taint
Taint clean rule (R2)	*TC* (taint-cleaner)	$f(m,n)=untainted,\ m\in VARS\wedge n\in TC$	Operations that reset the data field make tainted data de-taint
Taint propagate rule (R3)	*TR* (taint-relay)	$f(m,n)=level(m),\ m\in VARS\wedge level(m)<sensitive\wedge n\in TR$	Taint irrelevant operations pass on the security states, instead of changing them
Taint sink rule (R4)	*TS* (taint2sensitive)	$f(m,n)=sensitive,\ m\in VARS\wedge level(m)=tainted\wedge n\in TS$	Sensibilities occur when security operations refer to the tainted data
Sensitive propagate rule (R5)	*SR* (sensitive-relay)	$f(m,n)=level(m),\ m\in VARS\wedge level(m)=sensitive\wedge n\in SR$	Sensitive irrelevant operations pass on the security states, instead of changing them
Sensitive sink rule (R6)	*SV* (sensitive2vulnerable)	$f(m,n)=vulnerable,\ m\in VARS\wedge level(m)=sensitive\wedge n\in SV\wedge !filter(m,n)$	Vulnerabilities occur when operations with senstive data violate security constraints

6 种程序行为实体对变量的影响构成了各安全级间的转换,得到如图 4(b)所示的转换图,从初始状态⊥开始,清洁的数据经过 *TM* 变为污染的,*TS* 使用了污染数据产生敏感数据,敏感的数据在 *SV* 的作用下如果违背安全约束(*filter* 函数返回 False),将导致脆弱性的产生.与污点传播格模型[23]相比,4 种安全级细化了数据的污染状态和敏感状态的转化粒度,新增的 *sensitive* 安全级沿信息流逆向传播以标识出所有的危险整数操作,包括算术运算、赋值、位运算等,新增的 *tainted* 到 *sensitive* 的安全状态转换,将脆弱性的判定对象由污染信息流的目的地实体转移到信息流路径上的部分实体,新增的 *sensitive* 到 *vulnerable* 的安全状态转换,将涉及 *sensitive* 安全级整型变量的整数操作不满足静态安全约束作为脆弱性的判定规则.这样,便将脆弱性的判定由文献[23]的程序安全性操作引用 *tainted* 安全级变量推迟到 *sensitive* 安全级的约束求解之后,更为符合整数脆弱性产生的实际情况和整数漏洞的利用模式.同时,改进后的 *vulnerable* 安全级进一步降低了误报,只有当危险整数操作不满足安全约束(*filter* 函数的过滤)的情况下才有可能导致脆弱性.因此在我们的安全模型中,脆弱性的检查点(动态验证代码的插装点)是所有涉及 *vulnerable* 安全级整型变量的整数操作,即上述 *SV* 类型的程序行为实体.

由于整数漏洞的特点,基于信息流的检测方法需要着重关注污染信息流的路径和影响范围,因此,基于信息流的整数漏洞判定过程见定义 4 描述,信息流 *if* 各路径上实体的安全级均初始化为⊥,根据传播模式 *T* 和转换规则 *f*,如果有 *TM* 导致的污染数据没有被 *TC* 清除且经过 *TR* 的传递后流向了 *TS*,影响到程序的安全性产生敏感数据,并且敏感数据经 *SR* 的传递后被 *SV* 引用,就可判定其为潜在的整数漏洞.

定义 4(基于信息流的整数漏洞判定). 信息流 *if* 产生、传播了污染和敏感数据,导致潜在的整数漏洞,当且仅当$\exists p_1,\ldots,p_i,\ldots,p_j,\ldots,p_m,\ldots,p_k,\ldots,p_n\in obj(if),p_1,\ldots,p_i,\ldots,p_j,\ldots,p_m,\ldots,p_k,\ldots,p_n$ 在 *if* 中按传递顺序出现,$\exists m,n$,对$\forall i,j,k$,其中,$i<m,j<m,m<k<n$,使得 $f(p_1,p_2)\subseteq R1\wedge f(p_i,p_{i+1})\not\subset R2\wedge f(p_j,p_{j+1})\subseteq R3\wedge f(p_m,p_{m+1})\subseteq R4\wedge f(p_k,p_{k+1})\subseteq R5\wedge f(p_{n-1},p_n)\subseteq R6$.

下面以图 3 的漏洞实例为例,详细说明信息流安全模型是如何检测整数漏洞的.

按照定义 1,程序中存在 *png_get_IHDR*()到 *malloc*()的信息流动,共包括两条污染路径,分别是信息流:

- if_1: *png_get_IHDR*()→*width*→*img*→*xsize*→*bufsize*→*malloc*();
- if_2: *png_get_IHDR*()→*height*→*img*→*ysize*→*bufsize*→*malloc*().

初始状态下,所有的信息流实体的安全级均是⊥;*png_get_IHDR*()触发污染产生规则 R1,使 *width* 和 *height* 的安全级升级为 *tainted*;随后,污染状态沿信息流正向流动,按照污染传播规则 R3,根据定义 2 的安全级传播模式,*width* 和 *height* 的 *tainted* 安全级分别传递给 *img*→*xsize* 和 *img*→*ysize*,并最终传递给 *bufsize*,相应地使 *img*→*xsize*,*img*→*ysize* 和 *bufsize* 的安全级升级为 *tainted*;当 *tainted* 安全级的 *bufsize* 被内存分配函数 *malloc* 引用时,触发污染陷入规则 R4,*bufsize* 的安全级升级为 *sensitive*;随后,敏感状态沿信息流逆向流动,按照敏感传播规则 R5,根据定义 2 的安全级传播模式,*img*→*xsize*,*img*→*ysize*,*width* 和 *height* 的安全级同样由 *tainted* 升级为 *sensitive*;此时,共有 *img*→*xsize* 的赋值、*img*→*ysize* 的赋值和 *bufsize* 的乘法运算这 3 处整数操作涉及了 *sensitive*

安全级的整型变量,需要对它们进行静态安全约束求解,按照定义 3,并根据表 2 中 *filter*()的详细规约,前两个整数操作满足安全约束,不会构成整数漏洞,而 *bufsize* 的乘法运算在静态无法判定是否满足安全约束,保守判定为潜在的整数漏洞,需要插装验证代码以完成动态保护.

2.3 安全策略

2.3.1 污染描述符

制定污染数据的引入和安全使用策略,就是定义程序的可信边界(trusted-boundary),可信边界是系统不受外界影响的数据处理边界,边界的一端数据是不可信的,而另一端数据对特定操作是安全的.污染策略需要提供基于信息流检测整数漏洞的安全模型中污染数据的产生和安全操作,即对上述 6 种程序行为实体中的 *TM* 和 *TS* 进行细化.其中:*TM* 程序实体将污染数据引入到程序中,这些数据由用户控制,是污染数据的源头;*TS* 程序实体以污染数据作为输入,对其的操作影响到程序的安全性,是污染数据的敏感操作点.本文的工作基础便是提取出所有的污染源到污染敏感操作点的信息流路径.

本文采用描述符(descriptor)[19]的概念,将污染的引入和使用策略表示为一个由实体类型 *role*、污染数据的类型或敏感操作的类型 *type*、程序操作 *op* 以及操作数位置 *loc* 组成的四元组:

$$\langle role,type,op,loc\rangle|role\in ROLES,type\in TAINT\_TYPES\cup VULN\_TYPES,op\in ACTS,loc\in N,$$

其中,*ROLES* 是安全模型中程序行为实体的类型集合,包括 *TM* 和 *TS*.对于 *TM* 实体而言,*TAINT_TYPES* 定义了引入的污染数据的类型,可分为文件数据 *FILE_DATA*、用户终端输入数据 *USER_DATA*、网络数据 *NET_DATA* 和系统环境数据 *ENV_DATA* 这 4 类.对于 *TS* 实体而言,*VULN_TYPES* 定义了污染敏感操作的类型,主要包括缓冲区溢出 *BUFFER_OVERFLOW*、内存溢出 *MEMORY_OVERFLOW*、格式化串错误 *FORMAT_STRING*、路径遍历错误 *PATH_TRAVERSAL*、控制流劫持错误 *CONTROL_FLOW_HIJACK* 和恶意文件执行 *MALICIOUS_EXECUTE* 这 6 类.*ACTS* 是程序行为集合,包括函数调用、数组访问和条件分支判断等操作.程序行为实体并非单纯的函数操作或访问操作,而是函数和特定参数以及操作和特定数据的组合,*N* 为自然数集合,约定操作数的具体位置.

以 *char*fgets*(*char*restrict buf,int n,FILE*restrict fp*)和 *char*strncpy*(*char*destin,char*source,int maxlen*)为例,分别介绍 *TM* 和 *TS* 的污染描述符形式:*fgets* 的描述符为〈*TM,FILE_DATA,fgets*,0〉,表示为 *fgets* 函数可通过第 0 个参数 *buf* 将污染的文件数据引入到程序体内;*strncpy* 的描述符为〈*TS,BUFFER_OVERFLOW,strncpy*,1〉和〈*TS,BUFFER_OVERFLOW,strncpy*,2〉,表示污染数据若通过第 1 个或第 2 个参数被 *strncpy* 使用即有可能造成缓冲区溢出漏洞.

2.3.2 敏感数据的静态安全约束

图 4(b)中安全级转换规则 R6 借助值范围传播分析提供的整型变量的可能取值范围、类型信息和精度信息,对涉及 *sensitive* 安全级变量的整数操作进行静态安全约束求解,将不满足约束的敏感整型变量的安全级转换为 *vulnerable* 安全级,作为插装验证的对象.

本文扩展 RICH[3]的子类型规则(sub-typing rules),不仅约束了类型转换和算术运算操作,还对指针偏移、取反操作、移位操作等进行规约.表 2 列举针对涉及 *sensitive* 安全级整型变量的 *SV* 程序行为实体的详细安全约束信息.值范围传播分析可以提供整型变量的值范围、类型和精度信息,其中:

- *type* 表示整型变量的类型,包括 *int*8_*t*,*int*16_*t*,*int*32_*t* 和 *int*64_*t* 以及相应的无符号类型;
- *prec* 标识某整型类型的精度(precision);
- *sign* 标识某整型类型的符号位;
- *val* 记录某整型变量的取值范围;
- max()和 min()存放某整型类型的最大和最小可能取值.

借助这些信息,本文按照 *filter* 函数的安全约束对 *SV* 程序行为实体进行过滤,对不满足约束关系的整数操作进行规则 R6 的安全级转换,其中,*range-check*(*A*,*B*)判断 *B* 变量的值能否用 *A* 变量的类型表示.

Table 2 Safety rules for *SV* entity

表 2 *SV* 类型实体的安全约束

STMT (***SV***)	*filter*()	Comments
ASSIGN_EXPR: *A*=*B*; *A*=(*cast*)*B*;	if (*A.prec*<*B.prec*) *range-check*(); else if (*A.prec*=*A.prec*) *A.sign*==*B.sign*? True: *range-check*(); else *A.sign*<0 && *B.sign*>0? *range-check*(): True;	According to the precision and type of *rhs* and *lhs*, range checks are deployed
ADD_EXPR: *A*=*B*+*C*;	positive, *B.val*+*C.val*≤max(*A.type*)? True: False negative, *B.val*+*C.val*≥min(*A.type*)? True: False	Result of addition must fall into the range of its *type*
MULT_EXPR: *A*=*B***C*;	same *sign*, *B.val***C.val*≤max(*A.type*)? True: False diff *sign*, *B.val***C.val*≥min(*A.type*)? True: False	Result of multiplication must fall into the range of its *type*
SUB_EXPR: *A*=*B*–*C*;	*unsigned*, *B.val*≥*C.val*? True: False diff *sign*, min(*A.type*)≤*B.val*–*C.val*≤max(*A.type*)? True: False	For both *unsigned*, *B* must be bigger than *A*; For different *sign*, check like *ADD-EXPR*
SHIFT_EXPR: *A*>>*B* or *A*<<*B*	0≤*B.val*≤*A.type.size*? True: False	*B* must be positive and less than the length of *A*
NEG_EXPR: *A*=–*B*; *ABS_EXPR*: *A*=|*B*|;	*B.val*!=min(*A.type*)? True: False	*B* cannot be the minimum of its *type*
PTR_PLUS_EXPR	*offset.val*≥0? True: False	*Offset* cannot be negative

2.3.3 脆弱数据的动态验证

由于静态分析的不精确性,经过安全约束求解之后,大量涉及 *vulnerable* 安全级整型变量的整数操作会被判定为潜在整数漏洞.针对每一个潜在整数漏洞都做出脆弱性警告,以要求用户修正是不现实的.解决的方法便是静态分析阶段在源代码中插装(instrumentation)动态验证代码,实现脆弱性的实时保护.

插装的验证函数以定制的方式来实现,允许灵活的动态验证策略,以满足不同环境下的性能要求.本文根据IOC[4]中对不同插装方法的比较和 RICH[3]中对处理程序的讨论,插装原则为:

- 针对可能触发溢出的整数操作(*ADD_EXPR*,*MULT_EXPR* 和 *SUB_EXPR*),插装基于 CPU 标志位的后验测试[4],根据 OF,SF 和 UF 等标志位判断运算是否溢出[4].
- 针对可能造成符号错误和截断错误的整数操作(*ASSIGN_EXPR*),插装先验条件测试[4].
- 针对移位和取负等操作,同样插装先验条件测试.其中,先验条件测试的验证过程与 *filter* 函数基本相同,只不过是采用汇编指令实现.

此外,不满足验证条件的处理程序为抛出警告信息.

3 实 现

3.1 原型工具DRIVER(detect and run-time check integer-based vulnerabilities with information flow)

大多数编译器在词法、语法分析之后都有程序优化的过程,本文选择将信息流提取、污点分析、危险整数操作约束求解和插装整合进入 GCC 编译器的优化框架.GCC 是由 GNU 开发的、前端可处理 C 等多种语言并且可运行在多种硬件平台上的开源编程语言编译器.首先,不同语言编写的源文件经过前端的语法分析过程,以中间表示 GIMPLE 的形式传递给中端;随后,经过中端一系列的优化趟(pass),对 AST 进行不同层次的优化处理,例如创建控制流程图 CFG、创建函数调用图 call-graph、转换 SSA 形式、死代码消除等.同时,GCC 还提供大量的接口供开发者调用,方便开发者实现各自的优化趟;经过大量优化后,GIMPLE 转化成与硬件相关的中间表示 RTL,再经过相关的优化处理;最后,根据具体的硬件环境生成相应的可执行文件.

本文在 GCC 的编译环境上实现原型工具 DRIVER,实现框架如图 5 所示.信息流提取模块、约束求解模块和验证代码插装模块作为优化趟嵌入中间表示 GIMPLE 的优化趟链表中,按照 GCC 的优化管理机制(pass manager)规定的顺序,对 GIMPLE 进行遍历分析.首先,根据配置文件提供的污染描述符,信息流提取模块抽取出所有的以用户控制数据为起点,程序安全性操作点为终点的污染信息流集合 *taint-IFs*;接着,借助 GCC 内置的值范围分析结果,对危险整数操作进行安全约束求解;随后,根据配置文件提供的验证代码,插装模块对潜在整数漏洞进行指令级的插装;最后,经过 GCC 的后端处理,得到 DRIVER 插装过的可执行文件.

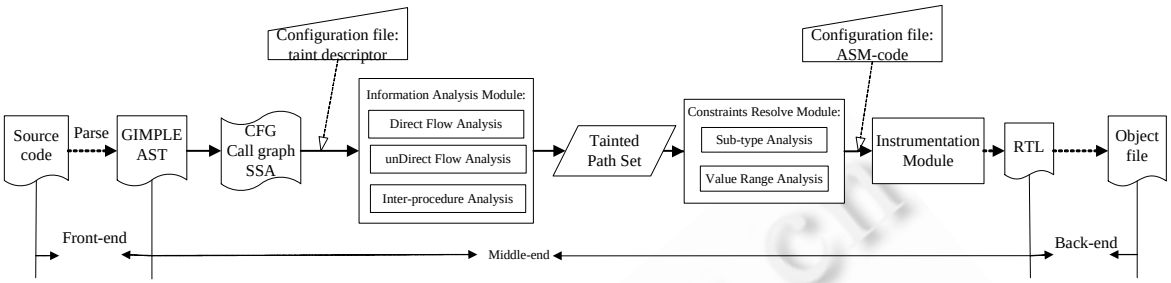


Fig.5 Architecture of DRIVER

图 5 基于信息流检测整数漏洞的实现框架

3.2 信息流提取模块

精确有效地抽取程序中的信息流是本文工作的基础.这里,基于静态单一赋值 SSA(static single assignment)中间表示,采用上下文敏感、过程间敏感、流敏感的信息流提取方法[23],根据污染描述符配置文件,抽取出程序中的污染路径.

首先,在编译器提供的 SSA 中间表示和控制流程图 CFG 等基础上,抽取出赋值、一元运算、二元运算、函数调用、函数返回、条件控制等基本操作的直接信息流.由于信息流具有传递性,多条直接流的中介和数据转移构成的信息传递路径,称为间接信息流.间接信息流以路径的起点和终点作为源和目的对象,其内部记录的路径点为信息流传递时经过的实体,而且任意两个相邻的路径点均可还原为直接信息流.本文采用深度优先遍历直接信息流图的方法计算任意两个信息流对象之间的所有可能路径,这里直接信息流图定义为 $DFG\langle N,E\rangle$,由于 SSA 的单一定值特性,*DFG* 是有向无环图,两个信息流识别对象间的所有间接信息流可通过计算二者在 *DFG* 中对应顶点间的所有可达路径[30]得到.

定义 5(直接信息流图). $DFG\langle N,E\rangle$,将信息流识别对象作为顶点,对象间的直接信息流动作为一条有向边,构成的有向图称为直接信息流图.其中,N 为信息流识别对象集 *OBJ* 的元素个数,E 为直接信息流的条数.

最后,在编译器提供的函数调用图和过程间优化机制的基础上,针对程序中的不同调用点进行过程间的信息流传递,以提取出程序中所有的 *taint-src*→*taint-sink* 信息流.

3.3 实现的不同分析粒度

3.3.1 编译单元的影响

为满足当前模块化编程的特点,编译器会将多个源文件划分到不同的编译单元(compilation-unit),每次只编译优化一个编译单元,最终将所有编译单元链接为目标文件.GCC 的 *unit-at-a-time* 编译模式就是这种模式,虽然该机制能极大减轻编译过程中对系统资源的需求,但给 DRIVER 的过程间信息流传递分析提出挑战.

如果一条过程间信息流的源节点和目的节点分布在不同的编译单元内,二者通过函数调用完成信息流动,那么在 *unit-at-a-time* 编译模式下,由于无法得到其他编译单元的语法树,DRIVER 难以将二者链接起来,进而会漏报一定数量的污染路径.

为解决此问题,本文提供两种分析粒度:

- 首先,进行完整的编译单元内的过程间分析,得到 *taint-src*→*taint-sink* 的污染信息流集合 *taint-IFs*,将这些污染信息流作为随后约束求解和插装的输入,称为 *taint* 粒度;
- 此外,污染数据可能通过函数形参流出到其他编译单元,这些信息流形如 *taint-src*→*prm*,构成可能污染信息流集合 *src-Ifs*;同时,函数形参也可能通过其他编译单元引入污染数据造成危险操作,这些信息流形如 *prm*→*taint-sink*,构成可能污染信息流集合 *prm-Ifs*.将这些可能污染信息流集合 *src-IFs* 和 *prm-IFs* 保守认为模糊的污染路径集合,作为随后约束求解和插装的输入,称这种分析粒度为 *prm* 粒度.

显然,*taint* 粒度精度高于 *prm* 粒度,但相应地,漏报率也高于 *prm* 粒度.

3.3.2 污染地址问题

如文献[25]中第 III 节 D 中所述,污染地址(taint address)问题是信息流污点分析一直存在的难点,可描述为:如果某内存单元是污染的,其地址需不需要标记为污染的?

以下面的程序片段为例,结构体 Person 有两个整型域 *age* 和 *weight*,已知在 *F*()第 6 行,指针变量 *P* 指向的 *age* 域标记为 *tainted*,若采用污染地址敏感模式,指针 *P* 也标记为 *tainted*,经过 *G*()的传递后,第 8 行的 *Q*→*age* 也是 *tainted*,并且此处进行敏感操作,进而产生一条污染路径⟨6,7,8⟩.若采用污染地址不敏感的模式,则该污染路径会被遗漏.然而,若第 8 行为对 *Q* 的 *weight* 域操作,在污染地址敏感的模式下会产生一条误报路径.为评估污染地址的敏感性对整数漏洞检测和静态分析的影响,本文实现两个分析粒度,分别为 *basic* 粒度和 *expand* 粒度.后者在提取直接信息流之后,对涉及内存地址引用的信息流做扩展,将 *a*[*i*]/*a*→*b*/**a*→*dest* 扩展产生新的 *a*→*dest* 的信息流.由第 4.1 节和第 4.3 节的实验可知,*expand* 分析粒度对漏洞检测的精度有极大的提高,但也增加了静态分析的负荷.

```
In function F():
  1 struct Person {
  2    int age;
  3    int weight;
  4 };
  5 struct Person*P=malloc…;
  6 P→age=taint-source-data;
  7 G(P);
In function G(struct Person*Q):
  8 sensitive-op on Q→age;
```

综上所述,由于考虑编译单元和污染地址敏感问题对检测精度的影响,本文实现的原型工具 DRIVER 共分为 4 个分析粒度,分别为 *basic-taint* 粒度、*basic-prm* 粒度、*expand-taint* 粒度和 *expand-prm* 粒度.

4 性能评价

本文选择一系列开源软件对 DRIVER 的漏洞检测能力、插装密度、污染地址敏感的影响、额外性能开销、新发现的漏洞这 5 个方面进行评估.由于 IntPatch[5]只识别可能造成 IO2BO 脆弱性的整数溢出漏洞,在插装密度和额外开销等方面不便对比,因此选择 RICH[3]进行多角度的对比实验.实验环境为:CPU Intel Core i5,主频 2.67GHz,内存 2GB,内核 linux-kernel-3.3.7,编译环境 GCC-4.3.3.

4.1 漏洞检测能力

为评估 DRIVER 的检测能力,首先选择 CVE 统计的 2008~2011 年最新的 93 个已知整数漏洞对其进行测试,涉及的开源软件包括 Linux Kernel,Apache,PHP,Python,GLIBC,GIMP,Jasper 等.需要说明的是,由于个别软件需要的编译环境在本文的实验环境下难以实现,对于这部分漏洞,以抽取的模型程序来代替,不过尽量保有其源程序的所有结构特征.

分析结果见表 3,第 1 列表示实验将从 4 类漏洞形式和总体来展示结果;第 2 列描述测试对象 93 个整数漏洞的分布情况,其中包括 65 个整数上溢、11 个整数下溢、16 个符号错误和 1 个截断错误;第 3 列分别介绍 DRIVER 的 4 种分析粒度的漏洞检测数量;第 4 列分别介绍它们的检测精度.从数据可以看出,*taint* 粒度的检测能力很低,约为 2%,说明程序中绝大多数污染路径的 *taint-source* 点和 *taint-sink* 点处于不同的编译单元内,这也是如今程序模块化编程普及的结果.*prm* 粒度的检测能力相对高,其中 *expand-prm* 的检测能力高达 92%,远高于 *basic-prm* 的检测精度.由此可以说明,程序中有很多漏洞的传递会涉及到数组或者结构体指针.

有 7 个整数漏洞没有被 *expand-prm* 粒度检测出,其中:

- CVE-2011-2511 的污染数据的传递过程中涉及了函数指针,DRIVER 无法有效处理;
- 由于 *expand* 版本仅对地址做单层扩展,对多级索引 *A*→*B*→*C* 没有处理,因此无法识别 CVE-2010-1411 和 CVE-2008-1721;
- 由于模块化编程的影响,污染路径的 *source* 点或 *sink* 点会被封装起来放在不同的编译单元里,因此无法识别 CVE-2010-4529,CVE-2009-3296,CVE-2011-0408 和 CVE-2009-2688.

Table 3 Results of testing DRIVER with real-world vulnerabilities

表 3 DRIVER 检测能力分析结果

	No.	No. of detected				Ratio of detection			
		Basic		Expand		Basic		Expand	
		Taint	Prm	Taint	Prm	Taint (%)	Prm (%)	Taint (%)	Prm (%)
Overflow	65	1	31	1	61	1.54	47.69	1.54	93.85
Underflow	11	1	3	1	9	9.09	27.27	9.09	81.82
Sign err	16	0	9	0	15	0.00	56.25	0.00	93.75
Truncate	1	0	1	0	1	0.00	100.00	0.00	100.00
Total	93	2	44	2	86	2.15	**47.31**	2.15	**92.47**

现将 DRIVER 与 RICH 在检测能力方面进行对比,按照 RICH 选择的测试对象进行分析,检测结果见表 4.DRIVER 能检测 9 个整数漏洞中的 8 个,检测能力与 RICH 基本相当,其中,samba 2.2.7a 的 *reply_nttrans*()漏洞无法检测的原因是污染路径的 *taint-source* 点被封装在另外的编译单元中;而列表中的最后一个漏洞 RICH 无法检测的原因是 RICH 不支持指针别名分析,无法获得用指针访问变量时的类型信息.由于 DRIVER 不需要详细的类型信息,因此可以检测出该漏洞.

Table 4 Comparison of detection ability between DRIVER and RICH

表 4 DRIVER 与 RICH 的检测能力对比

Program	Vulnerability	Kind (s)	Caught (DRIVER)	Caught (RICH)
samba 2.2.7a	statcache.c:206	Sign err	Yes	Yes
samba 2.2.7a	*reply_nttrans*() bug	OF	No	Yes
samba 2.2.7a	Directory ACL bug	OF	Yes	Yes
ProFTPD 1.2.10	mod_auth_unix.c:434	OF	Yes	Yes
Pine 4.55	Header parsing bug	OF	Yes	Yes
Mailutil-0.6 Imap4d	*fet_io*() bug	OF	Yes	Yes
PuTTY 0.53b	SFTP client bug	Sign err	Yes	Yes
mod_auth_radius 1.5.7	RADIUS reply bug	UF, sign err	Yes, Yes	Yes, Yes
GNU Radius 1.2	SNMP DoS	Sign err	Yes	No

4.2 插装密度

由表 3 可知,*expand-prm* 分析粒度下 DRVIER 的检测能力最高,这里通过表 5 描述该分析粒度下验证代码的插装密度.为方便比较,这里选取 RICH 中的 Apache 2.2.0,Samba 2.2.7a,ProFTPD 1.2.10,gzip 1.2.4 和 IntPatch 中的 libtiff 3.8.2,ming 0.4.2,faad2 2.7,dill0 2.0,gstreamer 0.8.5 作为分析对象.但由于 ProFTPD 1.2.10 的编译需要 2.4 内核,在本文的实验平台中无法正常编译,因此这里使用高版本的 ProFTPD 1.3.3d 代替.

Table 5 Results of instrumentations by *expand-prm* DRIVER

表 5 *expand-prm* 粒度下 DRIVER 的插装情况

Program	Version	SIZE (K)	*IFs*			Resolve information						
			taint-path	*src-path*	*prm-path*	*int-ops*	OF	UF	*Sign*	*Trunc*	Others	Total
Apache httpd	2.2.0	101	416	52	124 658	4 192	488	264	470	68	46	1 336
Samba	2.2.7a	189	6 140	127	196 598	4 743	1 271	250	618	56	2	2 197
ProFTPD	1.3.3d	112	419	53	35 313	1 027	128	74	153	8	5	368
gzip	1.2.4	7	74	4	305	161	28	14	36	0	3	81
libtiff	3.8.2	45	339	7	66 732	3 059	610	284	525	48	38	1 505
faad2	2.7	29	0	0	63 605	5 428	852	125	248	185	6	1 416
ming	0.4.2	78	4	0	31 282	2 314	573	71	219	51	14	928
dillo	2.0	13	391	0	14 220	508	87	79	85	1	4	256
gstreamer	0.8.5	50	256	2	50 116	1 398	205	108	142	68	12	535

表 5 的前 3 列分别描述了测试用例的名称、版本号和有效代码行数,第 4 列描述信息流路径的数量,最后一列详细介绍了各个测试用例的污染路径上整数操作的总数、对各个类别整数漏洞的插装个数和插装总和信息.由 *int-ops* 列的危险整数操作个数除以程序中所有的整数操作个数,得到危险操作占所有整数操作的比例约为 32%,即相比类型约束方法,本文的分析方法缩减约束求解对象的效果;由 total 列的插装个数除以 *int-ops* 列的

危险整数操作个数,得到可能引发漏洞的整数操作占危险整数操作的比例约为 40%,即,基于值范围传播分析可以避免 60%的危险整数操作的插装;由 SIZE 列的测试用例有效代码行数除以 total 列的插装个数,得到 DRIVER 的插装密度大约为每 92 行插装 1 次,远低于 RICH 的每 23 行插装 1 次.

4.3 污染地址敏感的影响

如第 3.3.2 节所述,考虑污染地址敏感问题虽能提高 DRIVER 的检测能力,但也会增加静态分析的负荷.图 6 描述了 *expand-prm* 粒度相比 *basic-prm* 粒度在直接信息流个数、间接信息流个数、污染路径个数、识别到的整数操作个数和插装点个数这 5 个维度上的倍数关系.平均来说,*expand-prm* 粒度在上述 5 个维度上分别是 *basic-prm* 粒度的 1.05 倍、2.04 倍、2.68 倍、2.12 倍和 2.03 倍.由此可见,污染地址敏感问题对静态分析的影响明显.

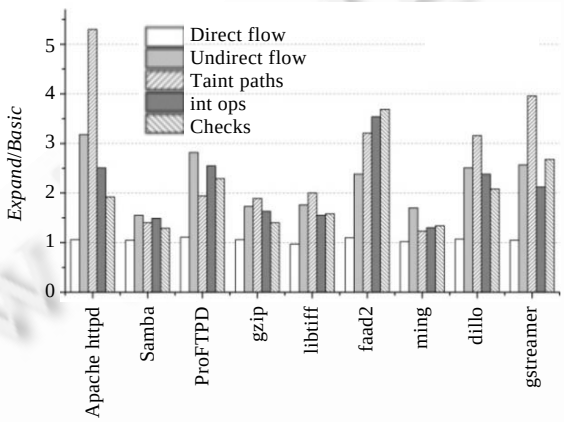


Fig.6 Influences on static analysis by taint-address sensitive analysis

图 6 地址敏感分析对静态分析的影响

4.4 性能开销

第 4.1 节~第 4.3 节着重介绍了 DRIVER 在静态分析阶段的实验情况,本节从动态运行的角度评估 DRIVER 的额外运行开销,测试对象包括 RICH 使用的测试程序和标准 SPEC2000 测试集.为确保实验数据的精确性,每个测试用例进行两组实验:第 1 组为未经 DRIVER 插装的可执行文件,第 2 组为经过 *expand-prm* 粒度的 DRIVER 插装后的可执行文件;同时,每组测试用例运行 5 次,取平均值作为运行时间;最后,再计算出经 *expand-prm* 粒度的 DRIVER 插装后的可执行文件的额外运行开销.

由于硬件实现平台的差异,对于 RICH 使用的 5 个测试程序,DRIVER 仅能对其中的 Apache Httpd 2.2.0 和 gzip 1.2.4 进行正常的测试.其中,对 Apache Httpd 运行测试用例 ab,DRIVER 的额外运行开销平均为 2.49%,远低于 RICH 的开销 8.18%;使用 gzip 解压 linux-2.6.15.tar.gz,DRIVER 的额外运行开销平均为 2.69%,高于 RICH 的开销 1.77%,这是因为 RICH 人为地删除针对部分误报的验证,其中涉及求模运算的良性整数溢出的循环操作,RICH 人为排除对这些误报的验证使开销由 300%降低到 1.77%,然而 DRIVER 并没有涉及人为处理.

仅测试两个程序不足以评估 DRIVER 的额外运行开销,因此本文继续对标准 SPEC 2000 中的 13 个 C 程序(由于 vortex 在当前系统硬件环境下对测试集处理时出现错误,这里排除对 vortex 的分析)进行测试,额外运行开销如图 7 所示,其中,额外开销最高的 mesa 不足 7%,最低的 gap 仅为 0.64%.平均而言,DRIVER 的开销约为 3%,低于 RICH 的 5%,这正是因为 DRIVER 仅对危险整数操作插装动态验证代码.

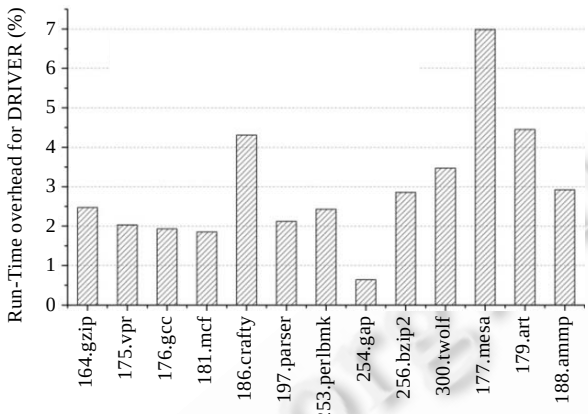


Fig.7 Performance overhead of SPEC 2000

图 7 SPEC 2000 额外运行开销

4.5 新发现的漏洞

DRIVER 在常见的开源软件中发现了一批潜在的未知漏洞,经过人工核查后,可以确定找到 11 个未知整数漏洞,见表 6.我们已将上述漏洞详细信息提交给 CVE 和 Secunia,正在等待确认,不便提供这些漏洞的更多信息.

Table 6 Zero-Day bugs detected by DRIVER

表 6 DRIVER 新发现的整数漏洞

Program	libtiff	xemacs	ImageMagick	cups	gnutls	rdesktop	samba
Version	4.0.1	21.4.22	6.7.8-1	1.6	3.0.21	1.7.1	3.6.6
Bugs#	1	1	2	1	2	3	1

4.6 小 结

综上所述,DRIVER 有较强的整数漏洞检测能力,在 *expand-prm* 分析粒度下检测精度高达为 92%.由表 4 可知,DRIVER 的检测能力与 RICH 相当.由第 4.2 节和第 4.4 节的实验可知,由于 DRIVER 采用减少静态分析对象数量的方法来降低开销,借助于信息流技术,仅仅对危险整数操作做约束求解和插装,其插装密度和额外运行开销明显小于 RICH,进一步验证了基于信息流检测整数漏洞方法的优势.由第 4.1 节和第 4.3 节的实验可知,考虑污染地址敏感问题的确能提高漏洞的检测精度,从 47.31%提高到 92.47%,但另一方面也导致静态分析的规模成倍增加;此外,DRIVER 发现了 11 个未知整数漏洞.

5 总 结

针对静态插装检测整数漏洞中的插装密度和额外运行开销的问题,本文通过深入探究整数漏洞的特征,提出了一种基于信息流检测整数漏洞的静态插装和验证方法.由于信息流可以追溯数据的产生、传递路径和使用,因此借助信息流,本文将整数操作与用户控制和程序安全操作匹配起来,将静态分析对象约减为污染信息流路径上的部分危险整数操作,从而达到降低插装密度和性能开销的目的.对原型系统的实验表明,基于信息流的插装验证方法能有效检测整数漏洞,性能开销较小(约为 3%),具有高精度、低开销、精确定位等特点.一些问题,如污染地址敏感问题、跨编译单元信息流传递问题、如何区分良性整数溢出等,还需要在接下来的工作中进一步研究.

References:

[1] Ahmad D. The rising threat of vulnerabilities due to integer errors. IEEE Security & Privacy, 2003,1(4):77−82. [doi: 10.1109/MSECP.2003.1219077]

[2] CVE statistics for integer vulnerabilities. http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer

[3] Brumley D, Chiueh T, Johnson R, Lin H, Song D. RICH: Automatically protecting against integer-based vulnerabilities. In: Proc. of the 14th Annual Network and Distributed System Security Symp (NDSS). San Diego: Internet Society, 2007.

[4] Dietz W, Li P, Regehr J, Adve V. Understanding integer overflow in C/C++. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). New Jersey: IEEE Press Piscataway, 2012. 760−770.

[5] Zhang C, Wang T, Wei TL, Chen Y, Zou W. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Proc. of the 15th European Conf. on Research in Computer Security. Berlin, Heidelberg: Springer-Verlag, 2010. 71−86. [doi: 10.1007/978-3-642-15497-3_5]

[6] Necula GC, McPeak S, Weimer W. CCured: Type safe retrofitting of legacy code. In: Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 2002. 128−139. [doi: 10.1145/503272.503286]

[7] SafeInt class. http://safeint.codeplex.com/

[8] Keaton D, Plum T, Seacord RC, Svoboda D, Volkovitsky A, Wilson T. As-if infinitely ranged integer mode. Technical Report, CMU/SEI-2009-TN-023, 2009.

[9] CERT. IntegerLib, a secure integer library. 2006. http://www.cert.org/secure-coding/IntegerLib.zip

[10] Molnar D, Li XC, Wagner DA. Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th USENIX Security Symp. San Diego: USENIX Association, 2009. 67−82.

[11] Cadar C, Dunbar D, Engler D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2008. 209−224.

[12] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. Exe: Automatically generating inputs of death. In: Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS). New York: ACM Press, 2006. 322−335. [doi: 10.1145/1180405.1180445]

[13] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. In: Proc. of the 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). New York: ACM Press, 2005. 263−272. [doi: 10.1145/1081706.1081750]

[14] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2005. 213−223. [doi: 10.1145/1065010.1065036]

[15] Sarkar D, Jagannathan M, Thiagarajan J, Venkatapathy R. Flow-Insensitive static analysis for detecting integer anomalies in programs. In: Proc. of the 25th Conf. on IASTED Int'l Multi-Conf.: Software Engineering. Anaheim: ACTA Press, 2007. 334−340.

[16] Wang TL, Wei T, Lin ZQ, Zou W. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proc. of the 16th Annual Network and Distributed System Security Symp. (NDSS). San Diego: Internet Society, 2009. 1−14.

[17] Pierce BC. Types and Programming Languages. Boston: The MIT Press, 2002.

[18] Jovanovic N, Kruegel C, Kirda E. Static analysis for detecting taint-style vulnerabilities in Web applications. Journal of Computer Security, 2010,18(5):861−907. [doi: 10.3233%2fJCS-2009-0385]

[19] Livshits VB, Lam MS. Finding security vulnerabilities in Java applications with static analysis. In: Proc. of the 14th Conf. on USENIX Security Symp. Berkeley: USENIX Association, 2005. 18−33.

[20] Chang R, Jiang G, Ivancic F, Sankaranarayanan S, Shmatikov V. Inputs of coma static detection of denial-of-service vulnerabilities. In: Proc. of the 2009 22nd IEEE Computer Security Foundations Symp. Washington: IEEE Computer Society, 2009. 186−199. [doi: 10.1109/CSF.2009.13]

[21] Liu Y, Milanova A. Static analysis for inference of explicit information flow. In: Proc. of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE). New York: ACM Press, 2009. 50−56. [doi: 10.1145/1512475.1512486]

[22] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In: Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS). San Diego: Internet Society, 2005.

[23] Huang Q, Zeng QK. Taint propagation analysis and dynamic verification with information flow policy. Ruan Jian Xue Bao/Journal of Software, 2011,22(9): 2036−2048 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3874.htm [doi: 10.3724/SP.J.1001.2011.03874]

[24] CUPS integer overflow vulnerability. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1722

[25] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In: Proc. of the 2010 IEEE Symp. on Security and Privacy. 2010. 317−331. [doi: 10.1109/SP.2010.26]

[26] Xu W, Bhatkar S, Sekar R. Taint-Enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proc. of the 15th USENIX Security Symp. Berkeley: USENIX Association, 2006. 121−136.

[27] Simon A. Value-Range analysis of c programs: Towards proving the absence of buffer overflow vulnerabilities. Springer-Verlag, 2008.

[28] Zitser M. Securing software an evaluation of static source code analyzers [MS. Thesis]. Boston: Massachusetts Institute of Technology, 2003.

[29] Dening DE. A lattice model of secure information flow. Communications of the ACM, 1976,19(5):236−243. [doi: 10.1145/360051.360056]

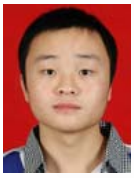
[30] Reps T. Porgram analysis via graph reachability. In: Proc. of the 1997 Int'l Symp. on Logic Programming. Cambridge: MIT Press, 1997. 5−19.

附中文参考文献:

[23] 黄强,曾庆凯.基于信息流策略的污点传播分析及动态验证.软件学报,2011,22(9):2036−2048. http://www.jos.org.cn/1000-9825/3874.htm [doi: 10.3724/ SP.J.1001.2011.03874]

孙浩(1987－),男,山东枣庄人,博士生,主要研究领域为信息安全,程序分析.
E-mail: shqking@gmail.com

曾庆凯(1963－),男,博士,教授,博士生导师,主要研究领域为信息安全,分布计算.
E-mail: zqk@nju.edu.cn



李会朋(1988－),男,硕士生,主要研究领域为信息安全,程序分析.
E-mail: lipe0530@126.com