DOI:10.3969/j.issn.1000－1565.2019.04.016

基于逆向计算的细粒度污点分析方法

屈雪晴，张圣昌
（河北大学 网络空间安全与计算机学院，河北 保定 071002）

摘 要：精度是污点分析的核心考虑因素.目前的污点分析方法，包括比特级污点分析的研究中，细粒度逻辑语义会导致精度缺失问题，而精度缺失直接致使“过度污点”现象.本文讨论了现有污点分析算法的局限性，阐述了污点传播过程中产生“过度污点”现象的原因，并提出一种基于逆向计算的细粒度污点分析方法，通过规定逆向计算规则，考虑语句的语义逻辑，推算污点传播策略.对未混淆代码和混淆代码分别进行污点分析的实验结果表明，相比于传统的污点分析算法，基于逆向计算的污点分析算法对混淆代码能够减少50％的代码冗余，有效地避免了污点的过度传播.

关键词：动态污点分析；反混淆代码；逆向计算；过度污点
中图分类号：TP309 **文献标志码**：A **文章编号**：1000－1565(2019)04－0437－07

Finer-grained taint analysis method based on reverse computing

QU Xueqing, ZHANG Shengchang
(School of Cyber Security and Computer, Hebei University, Baoding 071002, China)

Abstract: The precision is a core consideration in dynamic taint analysis.Current dynamic taint analysis algorithms, including previous studies on bit-level dynamic taint analysis, have more or less defects that can lead to serious lack of precision, and the lack of precision directly leads to over-tainting problem.This paper discusses the limitations of the traditional dynamic taint analysis algorithm, and explains the causes of over-tainting during the taint propagation, and proposes a reverse-computing finer-grained dynamic taint analysis algorithm to generate the strategy of taint propagation.Experiments using the deobfuscation tool show that the dynamic taint analysis algorithm proposed in this paper can reduce code redundancy by 50％ and significantly avoid the problem of over-tainting.

Key words: dynamic taint analysis; deobfuscation code; reverse computing; over-tainting

动态污点分析(DTA,dynamic taint analysis)是指在程序执行过程中将特定的数据(例如隐私数据，用户外部输入等)标记为污点，并跟踪其最终走向的技术.动态污点分析的应用领域十分广泛，包括隐私泄露检测、信息流控制、恶意软件分析以及代码反混淆.

由于上下文的数据依赖问题，传统的动态污点分析对污点传播策略抱有“鸵鸟哲学”，容易产生“过度污点”等问题，从而使得动态污点分析的性能开销极高.传统的动态污点分析工具会产生超过 30 倍左右的性能开销[1]，即使在基于 Pin[2] 插桩下，动态污点分析也会带来 6 倍的损耗.

针对动态污点分析的攻击也给分析带来了麻烦，例如编写恶意程序的攻击者不希望自己的恶意代码被

收稿日期：2018-09-13
基金项目：河北大学研究生创新项目(hbu2018ss58)
第一作者：屈雪晴 (1994—)，女，河北廊坊人，河北大学在读硕士研究生，主要从事分布计算和网络技术研究.
E-mail:724165288@qq.com

动态分析技术所检测[3-4],所以攻击者会将自己的代码进行混淆处理,将恶意代码分散到程序的各个部分,使得分析过程中产生性能爆炸的结果,这给动态污点分析带来了不便.

本文提出一种基于逆向计算的细粒度污点分析算法,通过指定基本的逆向计算规则,考虑语句的语义逻辑关系,推算污点传播策略.

1 相关工作

近年来,软件安全受到了广泛的关注,而动态污点分析则是检测软件漏洞的有效方法.国内外的学者在此领域也进行了较深入的研究.Valgrind 等[5]利用动态二进制检测(DBI)将污点分析与二进制程序执行分离,并分析了数据处理指令、算术逻辑指令的污点传播逻辑,以此提升了污点分析的效率.TaintCheck[6]在此基础之上将用户输入数据和网络数据进行了污点标记,此文详细讨论了污点传播策略,并成为了一套商用检测 x86 架构恶意代码的框架.

Dytan[7]是一个通用的污点分析框架,可以处理污点分析中潜在的数据流和控制流,Dytan 的特点是十分灵活,不需要运行时环境的限制.TaintDroid[8]是一个基于动态污点分析的 Android 端隐私泄露检测框架,TaintDroid 能够从变量级跟踪污点传播路径,将 Dalvik 解释器的变量语义跟污点类型对应,污点随着算术计算也跟着计算,从而避免了过度污点等问题,同时实现了文件级的污点跟踪,以确保动态污点分析的有效性.然而无论是 TaintCheck 还是 Dytan,或者 TaintDroid,都存在运行时开销过高的问题,在很多实时系统中,运行时高开销是绝对不允许的.

运行时开销过高的重要原因是"过度污点"现象[9].由于污点传播逻辑的欠缺,以及现代软件进行的代码混淆处理,为标准的污点分析算法带来了挑战[10].为了解决这个问题,目前针对污点传播分析的研究工作主要关注在设计有效的污点传播逻辑,以确保精确的污点传播分析[8].

近年来,污点分析技术已经用来推理和简化混淆代码,文献[11]通过优化控制流图来简化混淆代码,文献[12]在保留了逻辑语义的条件下,利用污点分析和符号执行相结合的方式对混淆代码还原.文献[13]使用了细粒度的污点分析方法,并对污点类型保留了逻辑语义,以确保污点的正确传播,只是上述 3 种方法的效果有限,对污点加入逻辑语义分析同样会带来存储开销和计算开销,影响了污点分析的效率.本文所设计的方法无需给污点添加逻辑语义,只需要一个比特位来表示污点标记即可,在最小开销下进行污点分析.

2 方法概述

2.1 动态污点分析

2.1.1 污点标记

本节将污点分析形式化表示,以阐述动态污点分析的基本思想.

定义 1(污点状态) 令 τ 表示污点状态集合,$\tau(v)=1$ 表示变量 v 被污点标记.

定义 2(污点分析) 令 $\xi(\tau,S)$表示指令段 S 在污点状态集 τ 上的污点分析过程.

本文所讨论的指令段 S 不包括控制转移语句,只包括基本的数据处理和逻辑计算语句.例如如下语句(1)

$$\xi(\tau,y:=x)=\tau[y\rightarrow\tau(x)], \tag{1}$$

其中 $\tau(x)$表示表达式的污点标记,[]操作符表示污点状态更新运算符.如果是一个新变量,则在 τ 污点状态集合中新建一个 $y\rightarrow\tau(x)$的映射关系,若 y 本就存在于 τ 中,则直接更新污点状态 $\tau(x)$.

污点标记是将敏感数据或外部输入进行记录并跟随数据流进行传播分析.初始化污点源由用户需求可自定义指定,例如系统调用,网络数据,用户输入等.污点标记通常使用 1 和 0 来表示,1 表示污点标记,0 表示正常数据.这种污点标记方法只需要一个比特位即可表示,所需空间小,却十分有效.

2.1.2 映射函数

为了理解映射函数,观察语句(2)

$$\xi(\tau, y := f(x)) = \tau[y \rightarrow \tau(x)]. \tag{2}$$

当指令表达式是一个复杂算术表达式时,污点该如何进行标记.映射函数定义了污点数据是使用何种方式传播的,也称之为传播策略.在传统的污点传播算法中,使用了"鸵鸟哲学",即无论 $f(x)$ 对 x 有何影响,都将 y 标记为污点,而这将会导致过度污点问题.假如

$$f(x) := x + 1,$$

则根据传统污点传播策略,亦可得出正确结果.而

$$f(x) := x - x,$$

则根据传统污点传播策略将无法得出满意的结果,因为此时 $y=0$,与 x 不存在相关性,利用逆向思维,无法使用 y 表示 x,表达式无法逆向推导,则 y 不应被标记为污点.针对此类问题,现阶段的解决方案是设定污点传播规则,然而设定规则不可能面面俱到,必有可绕过的方法.如表 1 所示,设定如下规则:

表 1　一条污点传播规则

Tab.1　A taint propagation rule

指令格式	指令语义	污点传播规则	解释
$sub\ v_1, v_2, v_3$	$v_1 \leftarrow v_2 - v_3$	$\tau(v_1)=0$	清空 v_1 的污点标记

这条传播策略可以通过语句(3)进行绕过

$$f(x) := x - z(z := x + 1). \tag{3}$$

通过将 z 定义为与 x 的线性相关,此时结果应该清除污点.因为 $f(x)=-1$,表达式不可逆.以这种方式绕过污点传播规则.所以本文使用了一种逆向的思想,简单的认为:若目标结果无法推导出源操作数,则称之为不可逆,污点也就不可传播.

2.1.3　粒度

粒度是表示污点标记的变量单位大小,显而易见粒度是影响污点分析准确性的重要因素.不同代码区域的污点粒度是不同的,粒度越精细,结果就越准确,在比特级粒度进行污点分析将会有更精准的检测效果.大量的研究显示,传统的字节或字级粒度的污点分析会导致"过度污点"问题,可是字节级粒度也会导致"性能爆炸".为了缓解这个问题,本文所采用的是可调整的细粒度方法.例如,在对内存的读写过程中,粒度以字节为单位进行污点标记,对于像 CPU 控制寄存器中的标志位,则使用比特级粒度进行污点标记.

2.2　逆向计算

逆向计算是指在污点传播路径中,对每一条指令在执行过程中进行逆向推算,推算的基本思想是,对当前指令进行分析,并根据目标变量和逆向规则获取到源操作数的值,假如源操作数被污点标记并且逆向计算成立,则将目标变量标记为污点,否则目标变量标记为正常.

2.2.1　基本定义

定义 3(逆向计算 Reverse Computing(RC))　$\psi=\{transition, rp, rs\}$,$\psi$ 为三元组,由 rp(指令地址),rs(指令语句描述),$transition$(逆向规则)组成.ψ 表示当一条指令在进行运算(逻辑运算或算术计算)后,可以通过相应的 $transition$ 推断出 rp 以及 rs.

定义 4(指令表达式 Instruction Expression(IE))　本方法讨论的指令为单源操作数:$y := (\lambda p : s)x$ $and\ ExsitV(\lambda p : s)=\varphi$

y 作为指令的目标操作数,x 是源操作数.$ExsitV(s)$则返回语句 s 中的变量集合.

$x, p, y \in Vars, s \in StExp, Vars \overset{def}{=}$ 变量集合,$StExp \overset{def}{=} Var \mid Z \mid StExp \quad op \quad StExp$ 表示一条指令语句,$op \overset{def}{=}$ 操作符.

对表达式的描述使用了 lambda 表达式形式.例如对于 $y := x - 1$,使用指令表达式为 $y := (\lambda p : p - 1)x$.或者使用函数名方式:$y := f(x), f \overset{def}{=} \lambda p : p - 1$.

定义 5(逆指令表达式 Reversible IE(RIE)) 若指令表达式 $y:=f(x)$,存在 $x:=f^{-1}(y)$ $ExsitV(f^{-1})=\varphi$,则称其指令可逆,其产生式即为逆指令表达式.RIE 的存在是不确定的,因为在指定数值的计算中,目标操作数有时无法直接推导出源操作数,例如指令 $y:=x-x$,通过计算 y 恒等于 0,不满足 RIE 的基本要求,故此指令不存在逆向计算.

2.2.2 **逆向规则**(transition)

逆向计算需要逆向规则引导产生 RIE,逆向规则可由用户自定义,或者通过公式推导得出.逆函数的推导规则是另外一门研究领域,由于本文所面临的指令都是基本运算,所以只提供基本的逆向规则(表 2)即可满足需求.

3 方法实现

3.1 污点传播框架

首先给出污点分析的源代码,见表 3.

表 2 逆向规则

Tab.2 Transition

1.$(\lambda x:x+k)^{-1}=\lambda x:x-k$	if	$x\notin ExsitV(k)$
2.$(\lambda x:k+x)^{-1}=\lambda x:x-k$	if	$x\notin ExsitV(k)$
3.$(\lambda x:x\times k)^{-1}=\lambda x:x\div k$	if	$x\notin ExsitV(k)\,\&\,y\neq 0$
4.$(\lambda x:k\times x)^{-1}=\lambda x:x\div k$	if	$x\notin ExsitV(k)\,\&\,y\neq 0$
5.$(\lambda x:x-k)^{-1}=\lambda x:x+k$	if	$x\notin ExsitV(k)$
6.$(\lambda x:k-x)^{-1}=\lambda x:k-x$	if	$x\notin ExsitV(k)$
7.$(\lambda x:x\div k)^{-1}=\lambda x:k\times x$	if	$x\notin ExsitV(k)\,\&\,k\neq 0$
8.$(\lambda x:k\div x)^{-1}=\lambda x:k\div x$	if	$x\notin ExsitV(k)\,\&\,x\neq 0$
9.$(\lambda x:x\oplus k)^{-1}=\lambda x:k\oplus x$	if	$x\notin ExsitV(k)$

表 3 基于逆向计算的污点分析算法源代码

Tab.3 Taint analysis algorithm source code based on reverse computing

算法 1:污点分析算法

```
Input:S(指令语句)
1.T = FindTaintSource(S)
2.TaintedVars = InitializeTaintedVars(T)
3.Markings = CreateMarkings(T)
4.while s = S.NextInstruction() do
5.        T = annotated(Markings,s)
6.        TaintedVars, Markings = Map(Markings,s)
7.endwhile
```

1)识别污点源:污点源的识别依据如下原则:任何可能被外部数据影响的变量都被视为对系统有潜在威胁,本文将用户输入和系统调用视作污点源,因为这些都是程序不可控信息.例如软件分析师对怀疑网络流量是威胁的来源,便可将网络数据流获取的系统调用视为污点源.

2)污点标记:本文使用混合粒度污点分析,例如对于内存中的污点传播,使用字节级粒度污点分析即可,而对 CPU 标志寄存器中的标志位,使用比特级粒度污点分析.为了减小开销,没有为污点标记分配类型,只使用了二进制表示法,1/0 表示是否是污点变量.

3)污点传播规则(Map):Map 函数设定了污点传播规则,判断一条指令 s 中,源操作数(假设其被标记为污点)是否将污点传播给目标操作数.本文同时利用了基本的污点传播规则和反向计算来设定 Map 函数,具体代码如表 4.

一般指令分为 2 类,移动指令、算术逻辑运算.

移动指令:此类指令大多为寄存器与寄存器之间或寄存器与内存之间的数据交换.此类语句的污点传播规则简单,本文使用文献[8]所提供的污点传播规则.

算术逻辑运算:此类指令是污点分析产生精度损失的主要根源,本文使用逆向计算来判别污点传播规则,利用逆向计算的结果来判断目标操作数是否应该被标记污点.

表 4　映射函数源代码

Tab.4　Source of Map funtion

```
算法 2:Map
Input:S(指令语句),Markings(污点标记)
1.Src = IdentifySource(s)
2.Dst = IdentifyDestination(s)
3.Switch s do:
4.      Case sdata handling and memory operations
5.          Dst.markings = Src.markings
6.      End
7.      Case sArithmetic and logic operations
8.          Dst.markings = ReverseComputing(s,Src,Src.markings)
9.      End
10.End
11.Return
```

本文依然保留了污点传播规则的自定义设置,用户可以自定义污点传播规则,以此对于简单的指令可以直接通过自定义规则来进行污点标记,无需对指令进行逆向计算,可以提高污点分析的效率.表 5 所示为常见的污点传播策略,其中 C 表示常数,R 代表返回函数,E 表示异常变量.

表 5　污点传播逻辑

Tab.5　Taint propagation Logic

指令格式	指令语义	污点传播规则	解释
move v_1,C	$v_1 \leftarrow C$	$\tau(v_1)=\varnothing$	清空 v_1 的污点标记
move v_1,v_2	$v_1 \leftarrow v_2$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
move v_1,v_2	$v_1 \leftarrow v_2$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
move v_1,R	$v_1 \leftarrow R$	$\tau(v_1)=\tau(R)$	返回值的标记赋值给 v_1
return v_1	$R \leftarrow v_1$	$\tau(R)=\tau(v_1)$	v_1 的标记赋值给返回值
move v_1,E	$v_1 \leftarrow E$	$\tau(v_1)=\tau(E)$	异常处理变量的标记赋值给 v_1
throw v_1	$E \leftarrow v_1$	$\tau(E)=\tau(v_1)$	将异常处理变量进行标记 v_1
add v_1,v_2,C	$v_1 \leftarrow v_2+C$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
sub v_1,v_2,C	$v_1 \leftarrow v_2-C$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
inc v_1	$v_1 \leftarrow v_2+1$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
dec v_1	$v_1 \leftarrow v_2-1$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给 v_1
neg v_1	$v_1 \leftarrow -v_1$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给
not v_1	$v_1 \leftarrow :v_1$	$\tau(v_1)=\tau(v_2)$	v_2 的标记赋值给

3.2　黑白名单机制

本文对污点分析进行优化,设置了黑名单和白名单机制,对常见的函数直接进行分析,如表 6 所示,列举了常见的函数.

表 6　黑白名单表

Tab.6　Blacklist and whitelist

类别	函数
白名单	strcmp,strncmp,memcmp,strlen,strchr,strstr,strbrk,strspn,qsort,rand,time,clock,ctime
黑名单	strcap,strncat,strcpy,strncpy,memcpy,memmove,strtok,atoi,itoa,abs,tolower,toupper

白名单中的函数可认为无污点传播,可直接跳过,用户可自定义设置白名单,随着白名单的条目增多,系统的效率会逐渐增加.而黑名单函数则认为存在污点传播途径,无需对函数内进行污点传播分析,直接对返回值进行污点标记.

4　实验评估

为了显示本文方法对混淆代码分析的有效性,提出了 2 个实验,第 1 个实验使用不同的动态污点分析测量被污点标记的变量数量,作为对比,对所测试程序使用了商业混淆工具 Code Virtualizer 对其进行代码混

淆,以比较本文的算法对混淆代码有良好的简化效用,避免了过度污点,本次实验测量的是污点分析的污点占有率.第 2 个实验测试不同污点分析方法对恶意程序的时间效率,以表明本文方法在时间上也有优化.

本文的实验均在 Intel core i7-6700 CPU 3.40GHz(8 CPUs)8GB 内存以及 Ubuntu18.04 Linux 环境下进行实验,并使用 3 个经典算法进行测试,快速排序(qsort),哈夫曼编码(huffman),动态规划算法(dp),以及一个恶意程序 CleanerPlus.exe,该文件来自公开的恶意代码共享库 malshare.

图 1 显示出每个程序的被污点标记指令的百分比.

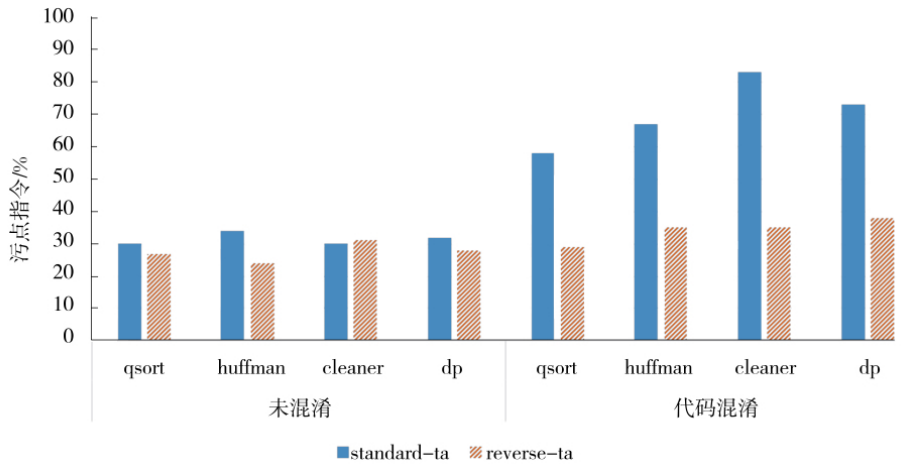

图 1 不同污点分析算法的污点标记数量

Fig.1 Quantitative data for various taint analysis approaches

其中未混淆表示原始程序,而代码混淆指的是通过 Code Virtualizer 工具进行混淆后的程序.如图 1 所示,本文的方法对原始指令的污点标记没有比标准污点分析更有优势,这是因为原始的程序代码直接简单,尤其是由编译器优化后更没有冗余代码.

而当进行代码混淆后,本文的方法体现出明显优势,传统的污点分析的污点标记均超过了 50%,产生了“过度污点”的现象.在对程序 clieanerPlus.exe 的进行分析的过程中,内存使用量达到 4 GB,这是由于 Code Virtualizer 的代码混淆,所分析的指令达到了 550 万条.而本文所提方法相对于混淆前的污点标记,混淆后污点标记量只提升了 1%~5%,这证明本文提出的方案在避免过度污点方面是可行的.

图 2 显示了各个程序所耗费的时间量,所耗时间是通过 5 次实验的算数平均值.

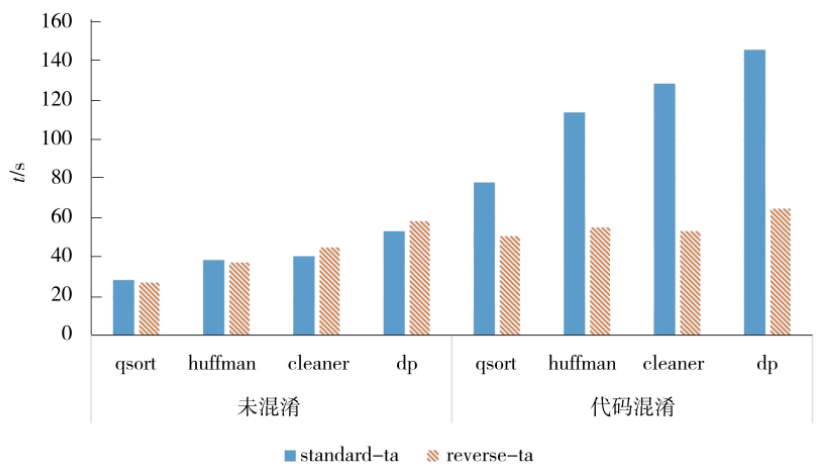

图 2 不同污点分析算法的所消耗的时间

Fig.2 Time for various taint analysis approaches

由图 2 所示,本文使用了多个策略对动态污点分析进行了优化,所以即使逆向计算消耗了一部分时间,但是通过优化,可以弥补这些计算的时间开销.在代码被混淆的状态下,本文提出的方案提升了动态污点分析的时间效率,这是因为逆向计算及时对无效污点进行了消除,节省了分析时间.

5　结束语

首先对动态污点分析进行了形式化描述,并分析了动态污点分析在混淆代码领域的弱势,提出基于逆向计算的污点分析方法.此方案以细粒度的方式进行污点传播,提高了污点分析的精度,以逆向计算作为污点传播策略有效地防止了污点的错误传播,提高了污点分析的有效性.

在与标准污点分析算法的对比实验中,使用了 3 个经典算法以及 1 个恶意二进制程序进行对比.本文提出的方案解决了代码混淆后的应用程序中"过度污点"问题,污点标记的语句数量在 4 次实验中平均减少了 50%.相比于标准污点分析算法,时间开销平均减少 55%.只是,本文提出的方案只考虑了显式数据流,而未来的工作是将此方案应用到隐式数据流中.

参　考　文　献:

[1] NEWSOME J, SONG D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]//In Proceedings of the 12th Network and Distributed System Security Symposium. 2005, 5: 3-4.

[2] KEMERLIS V P, PORTOKALIDIS G, JEE K, et al. Libdft:practical dynamic data flow tracking for commodity systems[Z].The 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, London, England, 2012. DOI: 10.1145/2151024.2151042.

[3] DYCHKA I, TEREIKOVSKYI I, TEREIKOVSKA L, et al. Deobfuscation of computer virus malware code with value state dependence graph[C]//International Conference on Computer Science, Engineering and Education Applications, Springer, Cham, 2018:370-379.DOI:10.1007/978-3-319-91008-6_37.

[4] SHENEAMER A, ROY S, KALITA J. A detection framework for semantic code clones and obfuscated code[J]. Expert Systems With Applications, 2018, 97: 405-420.DOI:10.1016/j.eswa.2017.12.040.

[5] NETHERCOTE N, SEWARD J. Valgrind:A framework for heavyweight dynamic binary instrumentation[J]. AcmSigplan Notices, 2007, 42(6):89-100. DOI:10.1145/1273442.1250746.

[6] CAI J, ZOU P, XIONG D, et al. A guided fuzzing approach for security testing of network protocol software[Z]. 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, 2015. DOI: 10.1109/ICSESS.2015.7339160.

[7] CLAUSE J, LI W, ORSO A. Dytan: a generic dynamic taint analysis framework[Z].The 2007 International Symposium on Software Testing and Analysis,ACM,New York, 2007. DOI: 10.1145/1273463.1273490.

[8] ENCK W, GILBERT P, CHUN B G, et al. TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones[J]. Transactions on Computer Systems, ACM,New York, 2014,32(2):1-29.DOI: 10.1145 / 2494522.

[9] DAI P, PAN Z, LI Y. A review of researching on dynamic taint analysis technique[Z].2018 3rd Joint International Information Technology, Mechanical and Electronic Engineering Conference, Paris, France,2018.. DOI:10.2991/jimec-18.2018.25.

[10] BANESCU S, COLLBERG C, GANESH V, et al. Code obfuscation against symbolic execution attacks[Z]. The 32nd Annual Conference on Computer Security Applications,ACM, New York, USA ,2016. DOI:10.1145/2991079.2991114.

[11] **王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. 软件学报**, 2017, 28(4):860-882.DOI: 10.13328/j.cnki.jos.005190.

[12] YADEGARI B, JOHANNESMEYER B, WHITELY B, et al. A generic approach to automatic deobfuscation of executable code[Z].IEEE Symposium on Security and Privacy, 2015. New York, USA. DOI:10.1109/SP.2015.47.

[13] YADEGARI B, DEBRAY S. Bit-level taint analysis[Z].IEEE International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, 2014.DOI:10.1109/scam.2014.43.

[14] XU Z, SHI C, CHENG C C, et al. A dynamic taint analysis tool for android app forensics[Z].2018 IEEE Security and Privacy Workshops (SPW), San Francisco, 2018.DOI:10.1109/SPW.2018.00031.

(责任编辑:孟素兰)