软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(6):1322−1339 [doi: 10.13328/j.cnki.jos.004592]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种具有时间多样性的虚拟机软件保护方法*

房鼎益[1,2], 赵 媛[1,2], 王怀军[2,3], 顾元祥[2,4], 许广莲[1,2]

[1](西北大学 信息科学与技术学院,陕西 西安 710127)
[2](西北大学-爱迪德信息安全联合实验室,陕西 西安 710127)
[3](西安理工大学 计算机科学与工程学院,陕西 西安 710048)
[4](爱迪德技术(北京)有限公司,北京 100125)
通讯作者: 房鼎益, E-mail: dyf@nwu.edu.cn

摘 要: 软件核心算法防逆向保护,是软件研发乃至软件产业发展的迫切需求,也是当前软件安全研究领域的热点之一.虚拟机软件保护作为一种保护强度高、商业应用广的技术,已被用于软件核心算法保护,并在很大程度上能够抵御攻击者的逆向分析.但这种保护方法难以抵御累积攻击,无法提供更加持久的保护.时间多样性是指一个软件在不同时间被执行时,执行路径不同,主要用于抵御累积攻击.将时间多样性与虚拟机软件保护相结合,提出了一种具有时间多样性的虚拟机软件保护方法,称为 TDVMP.在 TDVMP 中,通过构造多条相异的执行路径,使得被保护软件在不同次执行时,能够动态选取不同执行路径,从而极大地增加了攻击者进行累积的核心算法逆向分析攻击的难度.同时,对于 TDVMP 设计中的关键问题,比如多执行路径的构造与选择等进行了详细讨论.此外,提出了时间多样性保护效果的评价指标,并给出了其度量及计算方法.以所实现的原型系统为基础,通过一组具有一定实用价值的实例,对所提出的方法进行了测试、实验.结果表明,TDVMP 对于软件核心算法防逆向保护是有效且实用的.

关键词: 时间多样性;虚拟机软件保护;累积攻击;执行路径差异
中图法分类号: TP311

中文引用格式: 房鼎益,赵媛,王怀军,顾元祥,许广莲.一种具有时间多样性的虚拟机软件保护方法.软件学报,2015,26(6): 1322−1339. http://www.jos.org.cn/1000-9825/4592.htm
英文引用格式: Fang DY, Zhao Y, Wang HJ, Gu YX, Xu GL. Software protection based on virtual machine with time diversity. Ruan Jian Xue Bao/Journal of Software, 2015,26(6):1322−1339 (in Chinese). http://www.jos.org.cn/1000-9825/4592.htm

Software Protection Based on Virtual Machine with Time Diversity

FANG Ding-Yi[1,2], ZHAO Yuan[1,2], WANG Huai-Jun[2,3], GU Yuan-Xiang[2,4], XU Guang-Lian[1,2]

[1](School of Information Science and Technology, Northwest University, Xi'an 710127, China)
[2](NWU-Irdeto Network-Information Security Joint Laboratory (NISL), Xi'an 710127, China)
[3](School of Computer Science and Engineering, Xi'an University of Technology, Xi'an 710048, China)
[4](Irdeto Access Technology (Beijing) Co. Ltd., Beijing 100125, China)

Abstract: Anti-Reversing protection for persistent and high-insensitive software core algorithm has become an insistent demand for the research of software security and even for the whole software industry. Virtual machine based software protection has been widely used to protect the core algorithm from being reversed, but it is not sufficient for the current method to defend against cumulative attack and thus cannot provide long-term effective protection. Time diversity is used to fight against cumulative attack to allow software to execute along variant paths in different running time. A virtual machine based software protection method with time diversity, called TDVMP, is proposed in the paper. The key idea of the method is to construct multiple execution paths with equivalent semantics leading to

* 基金项目: 国家自然科学基金(61070176, 61170218, 61272461); 教育部高等学校博士学科点专项研究基金(20106101110018); 陕西省科技攻关计划(2011K06-07)

收稿时间: 2013-03-29; 修改时间: 2013-10-11; 定稿时间: 2014-03-07

© 中国科学院软件研究所 http://www.jos.org.cn

dynamically variant execution paths in running time. Main design issues of TDVMP, such as construction and selection of multiple execution paths, are discussed in detail. Furthermore, a metric named variation of execution paths to evaluate the effectiveness of time diversity is proposed, and the methods to measure and compute the metric are also presented. A prototype of TDVMP is implemented, and upon which the experiments are carried out with a set of practical use cases. Experiment results show that TDVMP is effective and applicable for core algorithm anti-reversing protection.

Key words: time diversity; VM-based software protection; cumulative attack; variation of execution path

软件中通常包含了核心算法、关键业务流程或用户身份等机密隐私及知识产权等关键信息,然而随着软件攻击与逆向技术的发展,破解与盗版的事件层出不穷,导致软件产业面临严重安全威胁.软件自身的安全性逐渐成为影响计算机软件行业发展的重要因素.由于发布后的软件处于无法预知的白盒攻击环境中,而且只要攻击者拥有足够的时间和资源,任何软件最终都会被破解[1].因此,急需探索能够提供持久保护的软件保护方法.

Collberg 于 2011 年提出:时间和空间上的多样性,是持久保护数字资产(包括软件)的必要条件之一[2,3].空间多样性是指同一个保护对象在多次保护后,不同版本的形态不同,通过控制版本的差异性来确保绝大部分软件处于安全状态,主要用于抵御共谋攻击[4].例如:文献[5]研究了软件多样性如何作为防御病毒传播的模型和仿真;文献[6]提出了使用自动化软件多样性来抵御内存错误入侵攻击;文献[7]提出了使用软件多样性方法来增强传感网的蠕虫攻击免疫能力;文献[8]提出了一种受限指令集的新技术,加强共谋攻击时软件多样性的应变能力.

时间多样性是指一个软件在不同时间被执行时,执行路径[9]不同.利用时间多样性可以抵御累积攻击[2].累积攻击是指通过多次逆向分析一个软件来累积攻击信息.累积攻击场景可描述为函数 $d=Attack(s,t,\xi)$,其中,s 代表被攻击软件,t 代表攻击次数,ξ代表 s 的时间多样性能力,d 代表逆向攻击进度.如图 1(a)所示,当$\xi=0$ 时,随着攻击次数的增加,逆向攻击进度的累积效果;如图 1(b)所示,当$\xi\neq 0$ 时,后一次的攻击不能有效利用前一次的攻击信息,逆向攻击进度难以累积.因此,对于同一个 s,累积攻击使得 d 与 t 正相关.时间多样性用于抵御累积攻击的实质是破坏 d 与 t 的正相关关系,ξ使得 s 在不同时间被执行时,执行路径不同,从而改变了 d 随 t 变化的正相关关系.

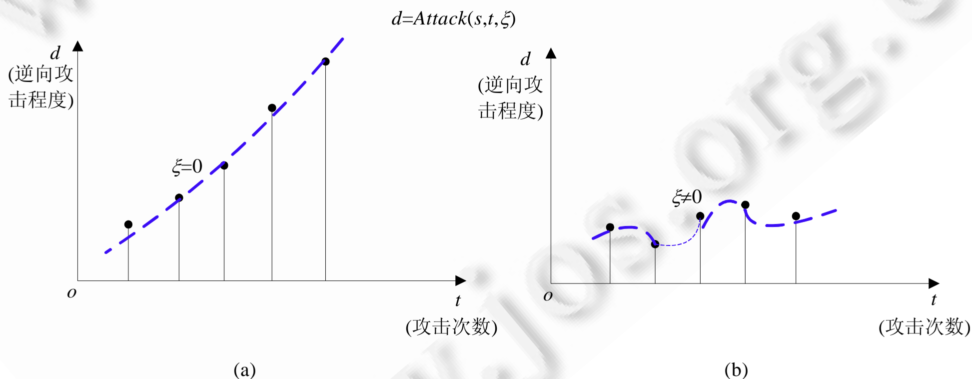

Fig.1 Cumulative attack scenario

图 1 累积攻击场景示意图

目前,虚拟机软件保护是一种保护强度大、商业应用广的软件保护技术,可提供对核心算法的保护[10−12].虚拟机解释器设计的复杂性及虚拟机保护的空间多样性,增加了攻击者动静态逆向分析的难度,在一定程度上保障了软件的安全性[13].然而,没有结合时间多样性的虚拟机软件保护难以抵御累积攻击,无法提供更加持久的保护.

本文针对虚拟机软件保护技术抗累积攻击差的问题,提出了一种具有时间多样性的虚拟机软件保护方法,称为 TDVMP(software protection based on virtual machine with time diversity),将时间多样性贯穿于 TDVMP 的不同阶段.采用代码克隆及 switch 块进行控制流混淆,能有效增强软件防逆向分析的安全性[14].然而,已有的代码克隆仅为简单的代码复制,且实际执行流程固定不变.在 TDVMP 中,代码克隆不仅是简单的复制,而是语义等

© 中国科学院软件研究所 http://www.jos.org.cn

价且形态不同的指令级变换;同时,Dispathcer-Handler 的调度结构不仅混淆了控制流,而且不同执行流程的代码形态不同.进一步结合时间多样性,通过构造多条相异的执行路径,使得被保护软件在不同次执行时,能够动态选取不同的执行路径.这种语义等价但执行路径各异的时间多样性软件保护,必将极大地增加累积攻击的难度.

本文的主要贡献在于提出了一种具有时间多样性的虚拟机软件保护方法 TDVMP,增强了虚拟机软件保护抗累积攻击的能力;提出了一种度量时间多样性效果的指标:“执行路径差异度 $\bar{d}_{path}$”及其计算方法.通过理论分析与实验,验证了具有时间多样性的虚拟机软件保护方法的可行性及其时间多样性效果.

1 相关工作

1.1 时间多样性

时间多样性思想被用于不同的软件保护场景,概括为如下 4 类.

(1) 跨平台移植

例如,文献[15]扩充了传统软件开发工具链,基于时间多样性设计了一款应用级虚拟机 strata,通过设计多套不同的虚拟指令集来支持被保护软件在不同平台运行时对不同指令集的选择,从而实现跨平台移植的目的;

(2) 在线安全更新

软件的完整功能必须结合服务器分发的代码块.每次更新时,将一些易受攻击的模块用形态不同的新代码块进行替换.例如,文献[16]提出了一种用于远程防篡改的具有多样性的软件保护方法,通过控制代码块更新周期小于软件的破解周期,保证软件处于安全状态;

(3) 抵御内存攻击

文献[17]采用的地址空间随机化技术,通过随机化栈地址及动态链接库基地址,实现不同次执行时实际的执行地址不同的效果,使得攻击者难以定位有效的内存数据;

(4) 防注入攻击

文献[18]采用指令集随机化技术来抵御注入攻击,通过对二进制代码进行动态随机加密,执行时对随机加密后的代码进行随机解密执行来抵御攻击者注入非法代码.

将时间多样性应用于软件保护时,具体的保护方法并不相同,但其本质都是使得被保护软件在不同次执行时能选取不同的执行路径.然而,将时间多样性应用于虚拟机软件保护中还需要进一步研究.

1.2 虚拟机软件保护技术

虚拟机软件保护技术[19−21]的原理是:将原始关键代码段(KeyCode)转换成虚拟指令(VI),并向被保护软件中嵌入虚拟机解释器(VM interpreter).虚拟机解释器由 4 部分组成:虚拟上下文(VMcontext)、处理函数集(Hdls)、分派函数(Dispatcher)和驱动数据(VMdata).其中:VMcontext 用于存储保护过程的中间结果(例如寄存器的值),以便执行时所需;Hdls 是由多个处理函数(handler)组成的集合,handler 是用来解释 VI 的指令片;VMdata 是由解释 VI 的 handler 的序号组成的数据序列;Dispatcher 根据 VMdata,调用 handler 执行.

虚拟机保护机制示例如图 2 所示.具体保护步骤如下:

Step 1. 对于输入的待保护软件 *P*,定位其 KeyCode 的起止范围,并根据 X86 指令与 VI 的映射关系,将 KeyCode 生成相应的 VI.如图 2 的 step 1 中,X86 指令 push edx 对应于 VI11,VI12,VI13.

Step 2. 构造 handler 集合 *Hdls*,并根据 VI 与 handler 的映射关系,将 VI 生成对应的 handler.如图 2 的 step 2 中,VI11,VI12,VI13 对应于 handler51,handler34,handler49,handler74,handler66.

Step 3. 将 handler 序号组织成为 VMdata.

Step 4. 构造 Dispatcher.

Step 5. 构造 VMcontext.

Step 6. 将 VMdata,Dispatcher,VMcontext,Hdls 组织成 VM Interpreter,并以新节形式嵌入被保护软件中.同时,将原始的 KeyCode 区域随机填充为花指令,并写入跳向 VM Interpreter 的 jmp 指令.输出被

© 中国科学院软件研究所 http://www.jos.org.cn

保护后的软件 P'.

在 P'执行时,当遇到 KeyCode 起始地址时,通过 jmp 指令跳向 VM Interpreter 开始解释执行.Dispatcher 读取一个 VMdata,计算后跳向 *Hdls* 中相应的 handler 执行.执行后重新返回 Dispatcher,读取下一个 VMdata.重复此过程至 VMdata 结束后返回,继续执行 KeyCode 的下一条指令.

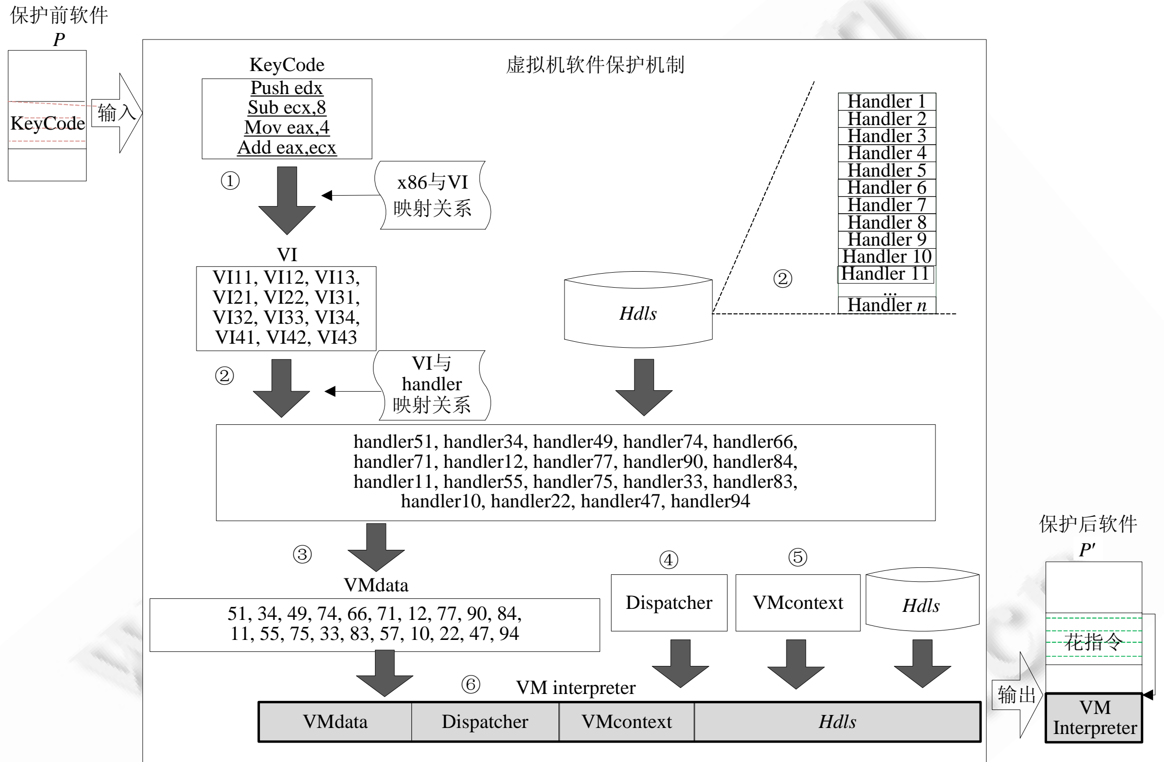

Fig.2 Mechanism of virtual machine protection

图 2 虚拟机软件保护机制

由此可见,虚拟机解释 KeyCode 的过程实际上是一定的 handler 按照一定的序列执行的过程.对一个攻击者而言,若要逆向分析一个被保护软件的 KeyCode,需要累积分析虚拟机所使用的 handler 及其不同次执行的 handler 序列.然而如前所述,由于被传统 VMP 保护后的 KeyCode 不同次执行时所用的 handler 序列都是固定的,因此会在很大程度上降低攻击者逆向分析的难度.

2 具有时间多样性的虚拟机软件保护方法

2.1 TDVMP方法概述

在不破坏虚拟机架构的情况下,TDVMP 将时间多样性与虚拟机保护过程中几个关键阶段(如生成虚拟指令、生成 VMdata、构造 *Hdls* 等)相结合.图 3 为 TDVMP 方法的保护过程.

一方面,TDVMP 通过 X86 指令混淆生成多个形态不同但语义等价的 handler 序列;另一方面,通过插入无效 handler 组合(详见第 3.1.2 节)及生成多套形态不同但语义等价的 handler 集合 *Hdls*,来增强多 handler 序列间的差异性.

在详细描述 TDVMP 的保护算法之前,先给出相关定义.

© 中国科学院软件研究所 http://www.jos.org.cn

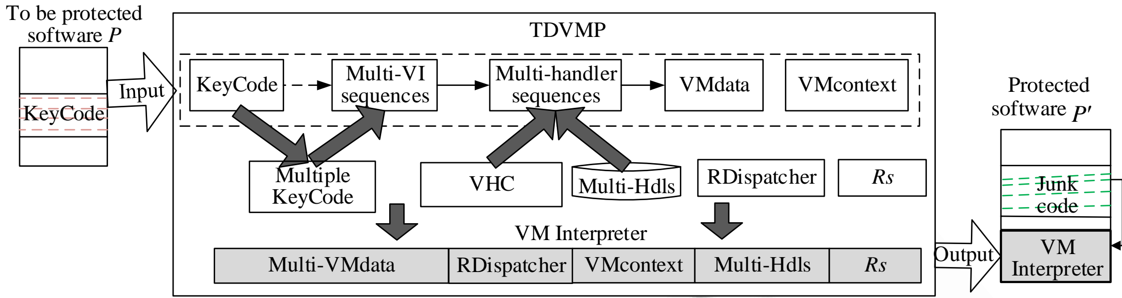

Fig.3 Protection process of TDVMP

图 3 TDVMP 方法保护过程

2.2 相关定义

定义 1(差异指令数 D、平均差异度 $\bar{d}$). 差异指令数指两段指令片 A_1,A_2 的相异指令条数,记为 $D(A_1,A_2)$.推而广之,m 段指令片的差异指令数 $D(A_1,A_2,\ldots,A_m)$表示任取两段指令片 $A_i,A_j(1\leqslant i,j\leqslant m)$的差异指令数的平均值.

$$D(A_1,A_2,\ldots,A_m)=(D(A_1,A_1)+D(A_1,A_2)+\ldots+D(A_{m-1},A_m)+D(A_m,A_m))/C_m^2 \tag{1}$$

平均差异度 $\bar{d}$ 指差异指令数在各指令片中所比例的平均值. $S_{A_1},S_{A_2},\ldots,S_{A_m}$ 表示指令片 $A_1,A_2,\ldots,A_m$ 包含的指令条数,则 $A_1,A_2,\ldots,A_m$ 的平均差异度计算如下:

$$\bar{d}=(D(A_1,A_2,\ldots,A_m)/S_{A_1}+D(A_1,A_2,\ldots,A_m)/S_{A_2}+\ldots+D(A_1,A_2,\ldots,A_m)/S_{A_m})/m \tag{2}$$

定义 2(一条指令多次混淆过程). 假设 I 为一条 X86 指令,c 为混淆次数,A_I 为指令 I 经过 c 次混淆后的指令片集合 $A_I=\{A_{I1},A_{I2},\ldots,A_{Ic}\}$.其中,$A_{I1}$ 表示指令 I 经过第 1 次混淆后对应的指令片,A_{I2} 表示指令 I 经过第 2 次混淆后对应的指令片,依次类推.那么,一条指令多次混淆过程可表示为 $A_I=obf(I,c)$,obf 为混淆函数.并且多次混淆后的指令片形态各异但语义等价,用符号⇔表示语义等价,则 $A_{I1}\Leftrightarrow A_{I2}\Leftrightarrow\ldots\Leftrightarrow A_{Ic}$.

定义 3(执行路径 *path*). 执行路径 *path* 指解释 KeyCode(假设包含 n 条指令)时所用的 handler 对应的指令执行序列,可表示为 $path:p_1\to p_2\to\ldots\to p_n$.其中,$p_j(1\leqslant j\leqslant n)$表示解释其中一条 X86 指令的执行路径,称为执行子路径,简称子路径.

通过对 KeyCode 进行 c 次混淆,构造 $m=c^n$ 条语义等价的用于解释 KeyCode 的执行路径(具体的构造过程详见保护算法),则其中第 i 条执行路径可表示为 $path_i:p_{i1}\to p_{i2}\to\ldots\to p_{in}$.那么,由 n 条子路径构成的 m 条执行路径便构成了 KeyCode 的全部执行路径 $PATH$,可表示为

$$PATH=\left\{\begin{matrix} path_1: p_{11}\to p_{12}\to\ldots\to p_{1n} \\ path_2: p_{21}\to p_{22}\to\ldots\to p_{2n} \\ \vdots \\ path_m: p_{m1}\to p_{m2}\to\ldots\to p_{mn} \end{matrix}\right\},1\leqslant m,n.$$

定义 4(子路径差异数 D_p、子路径平均差异度 $\bar{d}_p$ 与执行路径差异度 $\bar{d}_{path}$).

- 子路径差异数 D_p 指解释同一条 X86 指令的各子路径间的相异指令条数;
- 由定义 1 可知,解释 KeyCode 中第 1 条 X86 指令的 m 条子路径 $p_{11},p_{21},\ldots,p_{m1}$(即,$PATH$ 中虚线圈的第 1 列)的子路径平均差异度 $\bar{d}_{p_{m1}}$ 计算如下:

$$\bar{d}_{p_{m1}}=(D(p_{11},p_{21}\ldots,p_{m1})/S_{p_{11}}+D(p_{11},p_{21}\ldots,p_{m1})/S_{p_{21}}+\ldots+D(p_{11},p_{21}\ldots,p_{m1})/S_{p_{m1}})/m \tag{3}$$

  同理可计算分别解释其他 $n-1$ 条 X86 指令的 m 条子路径 $p_{12},p_{22},\ldots,p_{m2},p_{13},p_{23},\ldots,p_{m3},\ldots,p_{1n},p_{2n},\ldots,p_{mn}$ 的子路径平均差异度 $\bar{d}_{p_{m2}},\bar{d}_{p_{m3}},\ldots,\bar{d}_{p_{mn}}$,它们反映了 m 条执行路径 $path_1,path_2,\ldots,path_m$ 的局部差异性;
- 执行路径差异度 $\bar{d}_{path}$ 指所有子路径平均差异度 $\bar{d}_{p_{m2}},\bar{d}_{p_{m3}},\ldots,\bar{d}_{p_{mn}}$ 的平均值,它能够反映时间多样性效

果.因此,由 n 条子路径构成的 m 条执行路径的执行路径差异度 $\bar{d}_{path}$ 计算如下:

$$\bar{d}_{path}=(\bar{d}_{p_{m1}}+\bar{d}_{p_{m2}}+...+\bar{d}_{p_{mn}})/n \tag{4}$$

定义 5(时间多样性上限区间). 时间多样性上限区间指执行路径差异度的取值范围满足 $D(p_{1j},p_{2j},\ldots,p_{mj})\geqslant 1(1\leqslant j\leqslant n)$且 $D(p_{xj},p_{yj})\geqslant 1(x\neq y,1\leqslant x,y\leqslant m)$的时间多样性效果,它反映了抗累积攻击的能力.即:由 n 条子路径构成的 m 条执行路径中,用于解释同一条 X86 指令的各子路径间都存在差异指令,且其中任意两条子路径也都存在差异指令.

2.3 保护算法

在详细描述 TDVMP 保护算法之前,先对算法中使用的符号进行说明,如下表所示.

符号	描述
Rs	随机路径选择单元,用于选择不同的子路径
$Num(\cdot)$	计算包含指令数量的函数
$A_{I_{xc}}$	指令 I_x 第 c 次混淆对应的指令片
$hs(I_x)$	解释执行 $A_{I_{xc}}$ 用到的 handler 指令序列
HS	解释执行 A_I 用到的 handler 指令序列的集合
$HdlsSet$	对 Hdls 进行多次混淆后的多套 $Hdls$ 的集合
RDispatcher	改进的调度 handler 执行的分派单元

Step 1. 初始化 c=$CONSTANT$,n=Num(KeyCode).

Step 2. 对$\forall I_x\in$KeyCode($1\leqslant x\leqslant n$)进行 A_I=$obf(I,c)$操作,生成$\{A_{I_1},A_{I_2},...,A_{I_n}\}$.其中,

$$A_{I_1}=\{A_{I_{11}},A_{I_{12}},...,A_{I_{1c}}\},A_{I_2}=\{A_{I_{21}},A_{I_{22}},...,A_{I_{2c}}\},...,A_{I_n}=\{A_{I_{n1}},A_{I_{n2}},...,A_{I_{nc}}\}.$$

Step 3. 生成 A_{I_x} ($1\leqslant x\leqslant n$) 对应的 handler 指令序列:

$$HS=\left\{\begin{matrix} hs_{I_1}:hs(I_{11}),hs(I_{12}),...,hs(I_{1c})\\ hs_{I_2}:hs(I_{21}),hs(I_{22}),...,hs(I_{2c})\\ \vdots\\ hs_{I_n}:hs(I_{n1}),hs(I_{n2}),...,hs(I_{nc})\end{matrix}\right\},$$

其中,$hs(I_{x1})\Leftrightarrow hs(I_{x2})\Leftrightarrow\ldots\Leftrightarrow hs(I_{xc})$.

Step 4. 构造子路径及 c^n 条执行路径.子路径为 $hs(I_{ij})(1\leqslant i\leqslant n,1\leqslant j\leqslant c)$,一条执行路径由 n 条子路径构成,n 条子路径分别从 $hs_{I_1},hs_{I_2},...,hs_{I_n}$ 中选取.执行路径如下:

$$PATH=\left\{\begin{matrix} path_1:hs(I_{11})\to hs(I_{21})\to...\to hs(I_{n1})\\ path_2:hs(I_{12})\to hs(I_{22})\to...\to hs(I_{n2})\\ \vdots\\ path_{c^n}:hs(I_{1c})\to hs(I_{2c})\to...\to hs(I_{nc})\end{matrix}\right\}.$$

Step 5. 以子路径交叉方式将 VMdata 组织成一维结构.如下:

$$Rs_1hs(I_{11})hs(I_{12})...hs(I_{1c})Rs_2hs(I_{21})hs(I_{22})...hs(I_{2c})...Rs_i...Rs_{C^n}hs(I_{n1})hs(I_{n2})...hs(I_{nc}).$$

Step 6. 对$\forall handler_i\in Hdls$ 中的指令进行 A_I=$obf(I,c)$操作,其中,$1\leqslant i\leqslant k$,k 表示 $Hdls$ 中 handler 的个数.并且经过 c 次混淆后,得到:

$$HdlsSet=\left\{\begin{matrix} Hdls_1:handler_{11},handler_{12},...,handler_{1k}\\ Hdls_2:handler_{21},handler_{22},...,handler_{2k}\\ \vdots\\ Hdls_c:handler_{c1},handler_{c2},...,handler_{ck}\end{matrix}\right\}.$$

Step 7. 修改执行流程.

(1) Rs 初始化为选择第 1 个子路径,如 Rs_1 选择子路径 $hs(I_{11})$执行,Rs_2 初始选择子路径 $hs(I_{21})$执行;

© 中国科学院软件研究所 http://www.jos.org.cn

(2) 控制每个子路径 $hs(I_{ij})(1\leqslant i\leqslant n,1\leqslant j\leqslant c)$执行后跳回相邻的下一个 *Rs*,如 $hs(I_{12})$执行后需跳回 Rs_2 进行下一条子路径的选择;

(3) 将虚拟机解释器执行入口点指向 RDispatcher.

Step 8. 构造 VMcontext 和 RDispatcher,并与 *Rs*,VMdata,*HdlsSet* 一起嵌入被保护软件.

由第 1.2 节的研究可知:虚拟机软件保护中 KeyCode 与 handler 序列之间存在一对一关系,handler 序列以及 *Hdls* 与执行路径间存在多对多的关系.因此,TDVMP 通过构造多条 Handler 指令序列来构造多条相异的执行路径及生成多套 *Hdls* 来增强差异效果,并通过 *Rs* 实现被保护软件执行时动态子路径选择是有效的.其中,多执行路径的构造及随机路径选择单元 *Rs* 的设计是 TDVMP 实现中的关键,下一节将详细讨论.

3 TDVMP 中的关键技术

3.1 多执行路径构造

3.1.1 X86 指令混淆

如图 4 为通过 X86 混淆形成多 handler 序列的示意,其中略去生成虚拟指令(VI)的过程:

Step 1. 对 KeyCode 中的每条指令都进行如图 4 所示的混淆操作,生成多套等价的 KeyCode;

Step 2. 按照原始 KeyCode 的指令粒度,生成混淆后的每套 KeyCode 对应的 VI 片;

Step 3. 根据 VI 与 handler 的映射关系,将每个 VI 片映射为不同的 handler 序列.

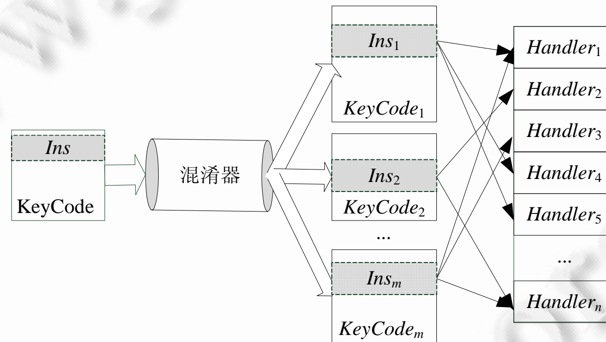

Fig.4 Multi-Sequences by X86 obfuscation

图 4 X86 混淆示意图

由图 4 可知,一条原始 X86 指令 *Ins* 经过 *m* 次混淆后,生成了 *m* 组语义等价但形态各异的 X86 指令片, $Ins\Leftrightarrow Ins_1\Leftrightarrow Ins_2\Leftrightarrow\ldots\Leftrightarrow Ins_m$.它们分别对应为 *m* 组语义等价但形态各异的 handler 序列,即,解释执行指令 *Ins* 的 *m* 条子路径, $handler_1\ handler_3\ handler_n\Leftrightarrow handler_2\ handler_n\Leftrightarrow\ldots\Leftrightarrow handler_1\ handler_4\ handler_5$.

3.1.2 无效 handler 组合

本文讨论的虚拟机软件保护技术中,handler 的设计是基于堆栈操作的,其中一些 handler 组合起来执行的效果等价于 nop 操作,这样的 handler 组合被称为无效 handler 组合(void handler combination,简称 VHC).

TDVMP 中设计了一种用来控制 VHC 插入的虚拟指令,称为 vhc.在生成虚拟指令的过程中,根据用户设置,将 vhc 插入到 VI 中.图 5 为用户设置 2~5(2~5 表示每隔 2~5 个 VI 便插入一个 vhc)情况下,插入 vhc 的示意图.

由图 5 可知:

第 1 个 vhc 被解析为 $handler_4\ handler_7\ handler_{11}$;

第 2 个 vhc 被解析为 $handler_9\ handler_{12}\ handler_4\ handler_7\ handler_{11}$;

第 3 个 vhc 被解析为 $handler_1\ handler_8\ handler_n$.

其中,第 2 个 vhc 对应的 handler 序列为两个原子 VHC: $handler_9\ handler_{12}$ 与 $handler_4\ handler_7\ handler_{11}$ 的叠加组合.

© 中国科学院软件研究所 http://www.jos.org.cn

一方面,无效 handler 组合(VHC)的插入,不会影响 handler 序列的功能;另一方面,不同 VHC 的随机插入,增强了不同 handler 序列间的差异性.

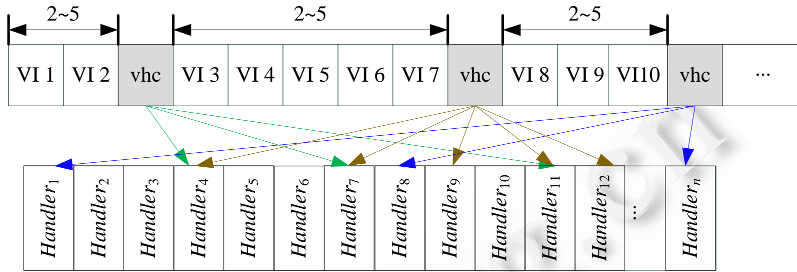

Fig.5 Multi-Sequences by inserting VHC

图 5 VHC 示意图

3.1.3 多 handler 集

将由 m 个 handler 组成的 *Hdls* 中的每个 handler 进行多次混淆(如图 4 所示),形成多套形态不同但语义等价的 *Hdls*.记为 $Hdls_1,Hdls_2,\dots,Hdls_n$.混淆过程满足以下条件:

条件 1. $Hdls_1,Hdls_2,\dots,Hdls_n$ 的 handler 数量相同;

条件 2. $Hdls_1(i)\Leftrightarrow Hdls_2(i)\Leftrightarrow\dots\Leftrightarrow Hdls_n(i)(1\leqslant i\leqslant m)$.

因此,被保护软件执行时,同一序号的 handler 从任意一套 *Hdls* 中选取均不影响 handler 序列的功能,但多套不同 *Hdls* 的存在,增强了不同 handler 序列间的差异性.

3.2 随机路径选择单元

由定义 5 可知,*Rs* 在虚拟机解释器执行时,完成如下功能:

① 控制对 *Hdls* 的随机选取;

② 控制对子路径的随机选取;

③ 在每一次随机选取后,调整执行流程,确保不破坏保护前后的语义.

Rs 由 *Hdls* 选择子单元,子路径选择子单元与执行流程调整子单元组成.TDVMP 中,*Hdls* 选择子单元集成于 RDispatcher 中,子路径选择子单元与执行流程调整子单元分别实例化为 Selector 与 Adjuster.Selector 是专门用来实现随机选取 handler 执行序列和决定是否执行 VHC 的特殊 handler.Adjuster 是专门用来实现运行时动态跨越 VMdata 的特殊 handler.它们的伪代码描述如图 6 所示.

RDispatcher	Selector	Adjuster
1. Read VMdata	1. Read VMdata:: $OFFSET_1$	
2. Decryptor VMdata	2. Decryptor VMdata	
3. Generator RadomSeed	3. Generator RadomSeed	
4. Compute *Rseed*	4. Compute *choice*	1. Read VMdata:: $OFFSET_2$
5. IF *Rseed*=0	5. IF *choice*=1	2. Decryptor VMdata
6. Choose $Hdls_1$	6. GO TO *L*	3. ADD EIP, *OFFSET*
7. ELSE	7. ADD EIP, *OFFSET*	
8. Choose $Hdls_2$	8. *L*:	
9. Jmp handler	9. Continue	

Fig.6 Pseudocode of *Rs*

图 6 *Rs* 的伪代码描述

3.3 被保护软件的执行

假设对 KeyCode 与 *Hdls* 都进行两次混淆.KeyCode 包含 mov eax,ebx 与 pop edx 两条指令,则被 TDVMP 保护后,解释执行 KeyCode 用到的 handler 序列,如图 7 所示.

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn

Keycode	X86 obfuscation	Handler sequences	Sub-Paths
mov eax, ebx	push ebx pop eax	00 01 0a 00 01 12	$path_{11}$

	sub esp, 4
	mov [esp], ebx
	mov eax, [esp]
	add esp, 4
		73 04 1f 15 07 04 05 01 74
		00 01 0a 75 07 04 05 01 12
		75 0a 00 01 12
		73 04 1b 15 07 04 05 01 74
			$path_{12}$

pop edx	mov [esp], edx add esp, 4	00 01 0a 75 07 04 05 01 12 73 04 1b 15 07 04 05 01 74	$path_{21}$
	xchg [esp], edx add esp, 4	7d 79 79 15b 73 04 1b 15 07 04 05 01 74	$path_{22}$

Fig.7 Multi-Paths of KeyCode for example

图 7 关键代码段多路径示意图

被保护的 KeyCode 具有 4 条执行路径:$path_{11}$→$path_{21}$,$path_{11}$→$path_{22}$,$path_{12}$→$path_{21}$,$path_{12}$→$path_{22}$,其执行效果均等价于执行 KeyCode.

作为示例,如图 8 只截取了 mov eax,ebx 对应的部分 handler 序列作为 VMdata,其中,第 1 条子路径为 00 01 0a 00 01 12,第 2 条子路径为 73 04 1f 15 07 04 05 01 74 00 01 0a 75 07 04 05 01 1275 0a 00 01 1273 04 1b 15 07 04 05 01 74.$Hdls_1$ 与 $Hdls_2$ 是 $Hdls$ 两次混淆后的结果.

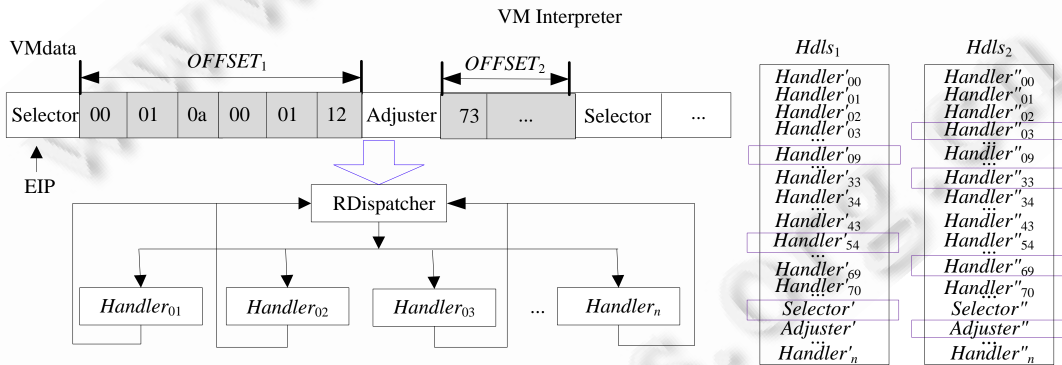

Fig.8 Execution of KeyCode

图 8 被保护软件执行示意

Step 1. 当遇到 KeyCode 时,执行流程转向 VM Interpreter;

Step 2. 执行 RDispatcher.读取 VMdata 得到 Selector 序号,随机从 $Hdls_1$ 或 $Hdls_2$ 中选取 Selector 去执行;

Step 3. 执行 Selector.读取 VMdata 得到 $OFFSET_1$,随机选取第 1 条或第 2 条子路径执行.如果执行第 1 条,则依次执行下一个 VMdata(00),执行完后,EIP 指向 Adjuster,进入 Step 4.如果执行第 2 条,则将当前 EIP 偏移 $OFFSET_1$ 后指向下一个 VMdata(73),依次执行完后,EIP 指向下一个 Selector;

Step 4. 执行 Adjuster.读取 VMdata 得到 $OFFSET_2$,将当前 EIP 直接偏移 $OFFSET_2$,指向下一个 Selector;

Step 5. 循环执行 Step 2~Step 4,直到读取完所有的 VMdata;

Step 6. 跳出 VM Interpreter,执行 KeyCode 的下一条指令.

在不破坏 VMP 基本架构的情况下,通过 RDispatcher,Selector 与 Adjuster 的配合,实现了 Rs 的所有功能,使得 TDVMP 很容易与现有虚拟机系统相结合,具有很好的实用性.

KeyCode 被不同次执行时,子路径序列的差异性便决定了不同次执行路径的差异性,也反映了时间多样性

© 中国科学院软件研究所 http://www.jos.org.cn

的效果.由于子路径是由逻辑地址连续的指令片组成,因此,本文采用差异指令数来度量子路径的差异性.同时,结合累积攻击场景,采用执行路径差异度来度量 TDVMP 的时间多样性效果.

下面从理论上分析执行路径差异度如何用于度量时间多样性效果及其计算方法.

4 时间多样性效果评测

由第 2.3 节中的 Step 4 可知:对于包含 n 条 X86 指令的 KeyCode,混淆 c 次后,可产生 c^n 条执行路径.如下:

$$PATH=\left\{\begin{matrix} path_1: hs(I_{11})\to hs(I_{21})\to ...\to hs(I_{n1}) \\ path_2: hs(I_{12})\to hs(I_{22})\to ...\to hs(I_{n2}) \\ \vdots \\ path_{c^n}: hs(I_{1c})\to hs(I_{2c})\to ...\to hs(I_{nc}) \end{matrix}\right\}.$$

根据公式(1):

$$D(A_1,A_2,...,A_m)=(D(A_1,A_1)+D(A_1,A_2)+...+D(A_{m-1},A_m)+D(A_m,A_m))/C_m^2=\overbrace{D(A_i,A_j)}^{C_m^2-m}\Big/C_m^2\ (1\leqslant i,j\leqslant m,i\neq j).$$

由定义 5 可知,$D(A_1,A_2,\ldots,A_m)$的时间多样性上限区间为:任取两段指令片 $A_i,A_j(1\leqslant i,j\leqslant m)$,其差异指令数均有 $D(A_i,A_j)\geqslant 1$.因此:

$$D(A_1,A_2,...,A_m)=\overbrace{D(A_i,A_j)}^{C_m^2-m}\Big/C_m^2\geqslant 1\times(C_m^2-m)\big/C_m^2=\left|\frac{m-3}{m-1}\right|.$$

根据公式(3):

$$\begin{aligned}\bar{d}_{hs_{1c}}&=D(hs(I_{11}),hs(I_{12}),...,hs(I_{1c}))/S_{hs(I_{11})}+...+D(hs(I_{11}),hs(I_{12}),...,hs(I_{1c}))/S_{hs(I_{1c})}\\&=\overbrace{D(hs(I_{1i}),hs(I_{1j}))}^{C_c^2-c}\times\left(\frac{1}{S_{hs(I_{11})}}+\frac{1}{S_{hs(I_{12})}}+...+\frac{1}{S_{hs(I_{1c})}}\right)\Big/C_c^2\\&\geqslant\left|\frac{c-3}{c-1}\right|\times\left(\frac{1}{S_{hs(I_{11})}}+\frac{1}{S_{hs(I_{12})}}+...+\frac{1}{S_{hs(I_{1c})}}\right).\end{aligned}$$

依此类推可得 $\bar{d}_{hs_{2c}},\bar{d}_{hs_{3c}},...,\bar{d}_{hs_{nc}}$.那么,由公式(4)可知,执行路径差异度 $\bar{d}_{PATH}$ 计算如下:

$$\bar{d}_{PATH}=(\bar{d}_{hs_{1c}}+\bar{d}_{hs_{2c}}+...+\bar{d}_{hs_{nc}})/n\geqslant\left|\frac{c-3}{n(c-1)}\right|\times\left(\frac{1}{S_{hs(I_{11})}}+...+\frac{1}{S_{hs(I_{1c})}}+\frac{1}{S_{hs(I_{21})}}+...+\frac{1}{S_{hs(I_{2c})}}+...+\frac{1}{S_{hs(I_{n1})}}+...+\frac{1}{S_{hs(I_{nc})}}\right).$$

推论 1. 对于包含 n 条 X86 指令的 KeyCode,经过 c 次混淆后产生的 c^n 条执行路径 *PATH*,其时间多样性的上限区间的最小值为执行路径差异度:

$$\bar{d}_{PATH}=\left|\frac{c-3}{n(c-1)}\right|\times\left(\frac{1}{S_{hs(I_{11})}}+...+\frac{1}{S_{hs(I_{1c})}}+\frac{1}{S_{hs(I_{21})}}+...+\frac{1}{S_{hs(I_{2c})}}+...+\frac{1}{S_{hs(I_{n1})}}+...+\frac{1}{S_{hs(I_{nc})}}\right).$$

推论 1 用来度量 TDVMP 的时间多样性效果,这将在 5 节中通过实验进一步验证.

5 分析与实验

5.1 攻击分析

软件攻击分为静态分析与动态分析,下面针对这两种分析方法对本文保护方法进行抗攻击分析.

(1) 针对静态分析

保护后,KeyCode 部分被填充为花指令,攻击者无法从中获取任何有用信息.要正确理解 KeyCode 的算法思想,必须分析 VM Interpreter 的整体设计思想,即,必须分析清楚 VMcontext,Dispatcher,handler,VMdata 等结构的含义及其相互配合关系.这显然需要软件攻击者付出大量的精力和时间,而且由于 handler 集合进行了变形混

© 中国科学院软件研究所 http://www.jos.org.cn

淆,VMdata 进行了动态加密,进一步增加了静态分析难度.显然,本文方法能够很好地防止攻击者进行静态分析.

(2) 针对动态分析

被保护软件执行过程中,读取加密后的 VMdata,通过 Dispatcher 解密并调度 selector,随机选取一个 handler 进行执行.重复此过程,直到解释完 KeyCode 中所有指令.软件攻击者在对保护后软件进行动态分析时,一般都要借助于逆向工具 OD,IDA 等,攻击者必须逐条指令分析,巨大的分析量无疑增加了分析的难度.此外,由于虚拟机结构的复杂性,使得攻击者无法在一次攻击中完全分析清楚整个架构和多套不同的 handler,必须进行多次攻击.而具有时间多样性的虚拟机使得软件每次执行时,执行路径均不相同,攻击者的攻击分析进度无法累积.因此,具有时间多样性的虚拟机软件保护方法更加有效地防止了软件攻击者的动态分析.

可见,静态分析和动态分析都不能帮助软件攻击者很好地逆向分析被保护软件,在很大程度上阻止了软件攻击者对软件核心算法的逆向分析.然而,Selector 的随机选择不可控,使得 Selector 本身的安全性成为一个安全瓶颈.不过,本文方法中,Selector 自身以 handler 的形式存在,与其他 handler 一起进行了变形切片乱序的保护,一定程度上增加了 Selector 的安全性.

下面将通过实验说明本文方法对时间多样性效果及性能的影响.

5.2 Nisl-vmp系统介绍

在对 TDVMP 深入研究的基础上,本文设计并实现了原型系统 Nisl-vmp,支持对 KeyCode 及 *Hdls* 进行两次混淆来产生时间多样性.其基本架构如图 9 所示.其中,X86 指令混淆、VHC 生成、handler 序列组织、多 handler 集构造、RDispatcher 及 *Rs* 构造模块是对 TDVMP 中核心部分的实现.

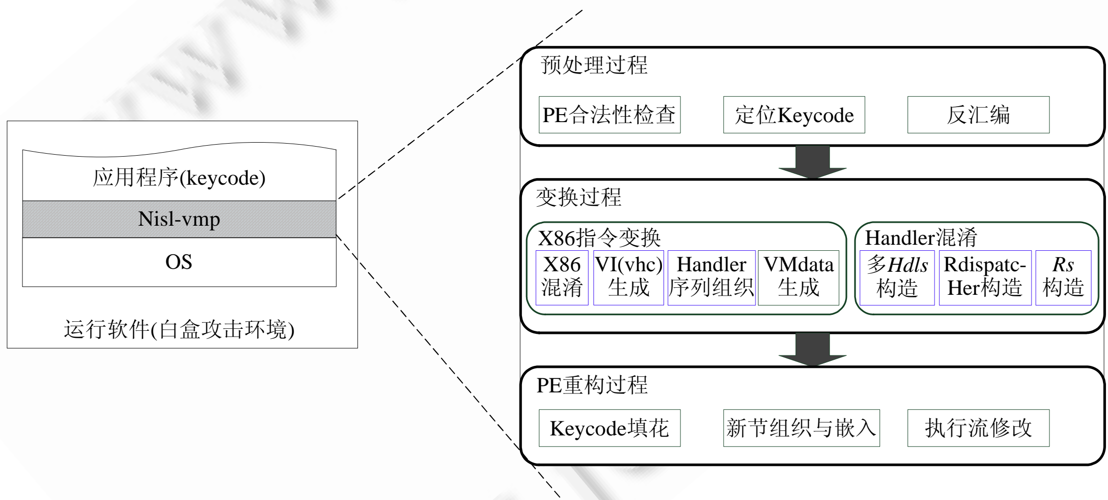

Fig.9 Framework of Nisl-vmp

图 9 Nisl-vmp 系统架构图

5.3 执行路径差异度计算

第 4 节中给出了执行路径差异度的计算过程,这里通过具体实例(以图 7 为例),说明根据执行路径、子路径、子路径差异数等信息来计算执行路径差异度.

① 执行路径=解释执行 KeyCode 所用的 handler 对应的指令执行序列.图 7 中执行路径有 4 条:

$$path_{11}\rightarrow path_{21}, path_{11}\rightarrow path_{22}, path_{12}\rightarrow path_{21}, path_{12}\rightarrow path_{22}.$$

② 子路径=解释执行一条 X86 指令的执行路径.图 7 中的子路径有 4 条:

$$path_{11}, path_{12}, path_{21}, path_{22}.$$

③ 子路径差异数=解释 KeyCode 中同一条 X86 指令的不同子路径间的差异指令数.

如图 10(a)是通过 IDC 脚本自动收集到的图 7 中 $path_{11}$ 对应的指令片.由于篇幅限制,图 10(b)截取了 $path_{12}$ 的前 3 个 handler(73,04,1f)对应的指令.其中,直线隔开的为一个 handler 的指令片.由于具有多套 *Hdls*,所以 $path_{11}$ 中两次用到的 00 或 01 号 handler 均不相同.

图 10(c)为通过 BCompare 对比工具提取图 10(a)、图 10(b)中差异指令的结果.

- $path_{11}$ 包括 41 条指令,$path_{12}$ 共包括 356 条指令(此处截取部分示意).由图 10(c)可知,子路径 $path_{11}$ 与 $path_{12}$ 差异指令数为 32;
- 同理,统计 $path_{21}$ 与 $path_{22}$,可得 $path_{21}$ 包括 96 条指令,$path_{22}$ 包括 132 条指令,它们的差异指令数为 47.

$$S_{path_{11}}=41, S_{path_{12}}=356, D(path_{11}, path_{12})=32,$$
$$S_{path_{21}}=96, S_{path_{22}}=132, D(path_{21}, path_{22})=47.$$

根据公式(3)可得:

$$\bar{d}_{p_{21}}=2\times\left(\frac{D(path_{11},path_{11},path_{12},path_{12})}{S_{path_{11}}}+\frac{D(path_{11},path_{11},path_{12},path_{12})}{S_{path_{12}}}\right)\Big/4=\left(\frac{32}{41}+\frac{32}{356}\right)\Big/2\approx 0.435,$$
$$\bar{d}_{p_{21}}=2\times\left(\frac{D(path_{21},path_{21},path_{22},path_{22})}{S_{path_{21}}}+\frac{D(path_{21},path_{21},path_{22},path_{22})}{S_{path_{22}}}\right)\Big/4=\left(\frac{47}{96}+\frac{47}{132}\right)\Big/2\approx 0.423.$$

根据公式(4)可得:

$$\bar{d}_{path}=(\bar{d}_{p_{21}}+\bar{d}_{p_{22}})/2\approx(0.435+0.423)/2=0.429.$$

```
lodsb
movzx    eax, al
lea      eax, dword ptr ds: [edi+eax*4]
sub      esp, 04
mov      dword ptr [esp], eax
----------------------------------------
mov      edx, dword ptr [esp]
add      esp, 04
----------------------------------------
push     dword ptr ds: [edx]
----------------------------------------
lodsb
movzx    eax, al
lea      eax, dword ptr ds: [edi+eax*4]
push     ebx
mov      ebx, esp
add      ebx, 04
sub      ebx, 04
xor      dword ptr [esp], ebx
xor      ebx, dword ptr [esp]
xor      dword ptr [esp], ebx
mov      esp, dword ptr [esp]
mov      dword ptr [esp], eax
sub      esp, 04
mov      dword ptr [esp], eax
----------------------------------------
mov      eax, edx
mov      eax, dword ptr [esp+4]
pushfd
mov      edx, eax
xchg     eax, dword ptr [esp]
xchg     eax, dword ptr [esp+4]
push     eax
push     dword ptr [esp+4]
pop      eax
pop      dword ptr [esp]
pop      eax
popfd
push     edx
mov      edx, esp
add      edx, 04
add      edx, 04
xchg     edx, dword ptr [esp]
pop      esp
----------------------------------------
pop      dword ptr ds: [edx]
```

(a)

```
sub      esp, 04
mov      dword ptr [esp], esp
push     ebx
mov      ebx, esp
add      ebx, 04
sub      ebx, 04
xchg     ebx, dword ptr [esp]
pop      esp
mov      dword ptr [esp], edx
mov      edx, 4
sub      esp, 04
mov      dword ptr [esp], edi
mov      edi, edx
add      dword ptr [esp+8], edi
pushfd
xchg     edi, dword ptr [esp]
xchg     edi, dword ptr [esp+4]
xchg     edi, dword ptr [esp]
pop      edi
popfd
pushfd
xchg     edx, dword ptr [esp]
xchg     edx, dword ptr [esp+4]
xchg     edx, dword ptr [esp]
pop      edx
popfd
----------------------------------------
lodsd
sub      esp, 04
mov      dword ptr [esp], eax
----------------------------------------
push     dword ptr ds:[edi+1C]
popfd
mov      eax, dword ptr [esp]
add      esp, 04
push     ecx
mov      ecx, eax
sub      dword ptr ss:[esp+4], ecx
pushfd
xchg     ecx, dword ptr ss:[esp]
xchg     ecx, dword ptr ss:[esp+4]
xchg     ecx, dword ptr ss:[esp]
pop      ecx
popfd
```

(b)

Fig.10 Variation X86 instructions of sub-paths

图 10 子路径指令片差异示意图

© 中国科学院软件研究所 http://www.jos.org.cn

```
lodsb
movzx   eax, al
lea     eax, dword ptr ds: [edi+eax*4]
sub     esp, 04
mov     dword ptr [esp], eax
mov     edx, dword ptr [esp]
add     esp, 04
push    dword ptr ds: [edx]
lodsb
movzx   eax, al
lea     eax, dword ptr ds: [edi+eax*4]
push    ebx
mov     ebx, esp
add     ebx, 04
sub     ebx, 04
xor     dword ptr [esp], ebx
xor     ebx, dword ptr [esp]
xor     dword ptr [esp], ebx
mov     esp, dword ptr [esp]
mov     dword ptr [esp], eax
sub     esp, 04
mov     dword ptr [esp], eax
mov     eax, edx
mov     eax, dword ptr [esp+4]
pushfd
mov     edx, eax
xchg    eax, dword ptr [esp]
xchg    eax, dword ptr [esp+4]
push    eax
push    dword ptr [esp+4]
pop     eax
pop     dword ptr [esp]
pop     eax
popfd
push    edx
mov     edx, esp
add     edx, 04
add     edx, 04
xchg    edx, dword ptr [esp]
pop     esp

pop     dword ptr ds: [edx]
```

```
sub     esp, 04
mov     dword ptr [esp], esp

push    ebx
mov     ebx, esp
add     ebx, 04
sub     ebx, 04
xchg    ebx, dword ptr [esp]
pop     esp

mov     dword ptr [esp], edx
mov     edx, 4
sub     esp, 04
mov     dword ptr [esp], edi
mov     edi, edx
add     dword ptr [esp+8], edi
pushfd

xchg    edi, dword ptr [esp]
xchg    edi, dword ptr [esp+4]

xchg    edi, dword ptr [esp]
pop     edi
popfd
pushfd
xchg    edx, dword ptr [esp]
xchg    edx, dword ptr [esp+4]

xchg    edx, dword ptr [esp]
pop     edx
popfd
lodsd
sub     esp, 04
mov     dword ptr [esp], eax
push    dword ptr ds: [edi+1C]
popfd
mov     eax, dword ptr [esp]
add     esp, 04
push    ecx
mov     ecx, eax
sub     dword ptr ss: [esp+4], ecx
pushfd
xchg    ecx, dword ptr ss: [esp]
xchg    ecx, dword ptr ss: [esp+4]
xchg    ecx, dword ptr ss: [esp]
pop     ecx
popfd
```

(c)

Fig.10 Variation X86 instructions of sub-paths (Contiuned)

图 10 子路径指令片差异示意图(续)

5.4 时间多样性效果分析

第 5.3 节及第 5.4 节中的实验硬件环境及 KeyCode 执行时间计算方法为:

① 实验硬件环境:Windows XP 系统;CPU:酷睿 2 双核;内存:DDR2 800 2GB;硬盘:7200 转/分钟.

② KeyCode 执行时间计算方法.

KeyCode 执行时间计算方法.

QueryPerformanceCounter(&t_1);	//开始计数
NISLVM_START	//Nisl-vmp 开始标记
KeyCode	
NISLVM_END	//Nisl-vmp 结束标记
QueryPerformanceCounter(&t_2);	//结束计数

```
llResult=t2.QuadPart−t1.QuadPart;                          //KeyCode 执行期间滴答数
QueryPerformanceFrequency(&freq);                          //CPU 内部定时器的时钟频率,即,每秒嘀哒声的个数
_i64toa(llResult*1000000/freq.QuadPart,szBuffer2,10);      //KeyCode 执行时间(ns)
```

为了测试 Nisl-vmp 的时间多样性效果,我们选取了 6 个常用的应用软件,详细描述见表 1.

Table 1 Description of experiment objects

表 1 实验对象描述

被保护软件	待保护 KeyCode 描述	KeyCode 指令数(条)	文件大小(字节)	KeyCode 执行时间(μs)
Hanio.exe	完美汉诺塔算法,并且输入盘子数目为 5	82	548 942	4 371
Ipmsg.exe	飞鸽中发送信息的算法	363	417 869	2 866
Calculator.exe	Windows 自带计算器程序的乘法操作算法	48	2 265 190	20
Csnake.exe	贪吃蛇游戏中,定位游戏界面的算法	60	233 563	0.000 26
Compress.exe	进行文件压缩的算法	110	229 447	1.334
Hviewer.exe	简易浏览器中,响应消息的算法	4	2 195 556	20

通过收集 handler 执行序列、统计执行路径数、一条执行路径包含的子路径数、子路径差异指令数及子路径总指令数等数据,来计算执行路径差异度.需要分别统计 82×2,363×2,48×2,60×2,110×2,4×2 条子路径的平均差异度.经统计计算,其子路径平均差异度及执行路径差异度分别如图 11、图 12 所示.

由图 11 可以看出,Nisl-vmp 保护后的软件的不同子路径平均差异度不相同,即,保护不同的 X86 指令产生的时间多样性效果并不相同,也验证了 TDVMP 保护算法的设计.分析如下:

① 针对不同的 X86 指令进行混淆时,Nisl-vmp 中采取基于 X86 指令变形引擎[22]进行混淆,混淆模板的差异直接导致 X86 指令混淆结果的差异;

② 不同无效 handler 组合(VHC)的自身差异及插入位置的不同,会导致不同子路径间的差异;

③ 多套 handler 集合(*Hdls*)带来的差异,Nisl-vmp 中对不同 *Hdls* 进行混淆,方法同情形①.

综上分析,TDVMP 中的 X86 指令混淆,VHC,多 Hdls 等技术均在一定程度上增强了 Nisl-vmp 的时间多样性效果.

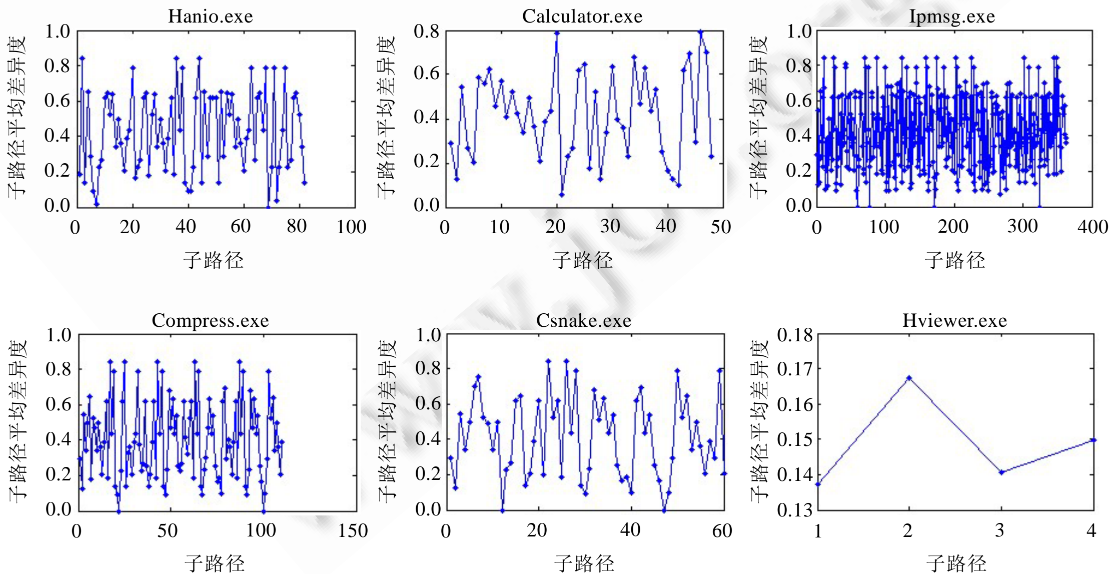

Fig.11 Average variation degree of sub-paths

图 11 子路径平均差异度实验结果

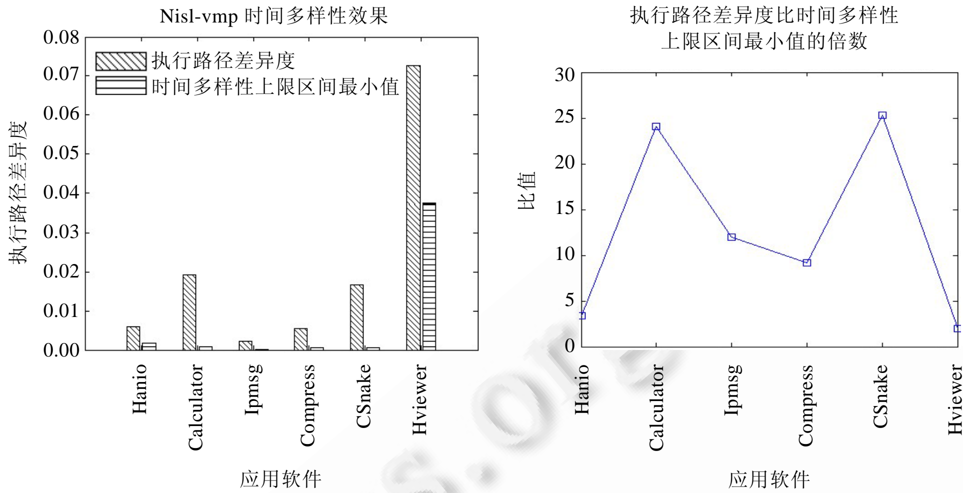

Fig.12 Variance degree of execution paths

图 12 执行路径差异度实验结果

图 12 表明:根据推论 1,Nisl-vmp 的时间多样性效果极大地超出了其时间多样性上限区间的最小值,即,这 6 个应用软件中的各个子路径间的差异指令远大于 1 条.说明 Nisl-vmp 的具有较好的时间多样性.

5.5 Nisl-vmp性能分析

在同样的实验环境下,对同样的测试实例(见表 1),分别用商用软件 Code Virtualizer1.3.1.0,VMProtect1.7.0 和我们开发的 Nisl-vmp 对其保护.其中,Code Virtualizer1.3.1.0 设置为最低保护力度,Virtual Machine Protector 1.7.0 设置为最快执行时间.

表 2 给出了各应用软件保护前后文件大小变化,表 3 给出了各应用软件保护前后 KeyCode 执行时间变化.

Table 2 Comparison of file size (byte)

表 2 保护前后文件大小变化(byte)

被保护软件	保护前(byte)	保护后(byte)		
		CV1.3.1.0	VMP1.7.0	Nisl-vmp
Hanio.exe	548 942	567 343	569 344	577 536
Ipmsg.exe	417 869	443 799	442 368	471 040
Calculator.exe	2 265 190	2 283 175	2 334 720	2 334 720
Csnake.exe	233 563	251 324	253 952	270 336
Compress.exe	229 447	249 050	249 856	262 144
Hviewer.exe	2 195 556	2 212 079	2 265 088	2 220 032

Table 3 Comparison of execution time of KeyCode (ms)

表 3 执行前后 KeyCode 执行时间变化(ms)

被保护软件	保护前(ms)	保护后(ms)		
		CV1.3.1.0	VMP1.7.0	Nisl-vmp
Hanio.exe	4 731	4 423	4 666	4 898
Ipmsg.exe	2 866	3 488	4 395	5 296
Calculator.exe	20	45	64	203
Csnake.exe	0.000 26	0.233	0.066	0.088
Compress.exe	1.334	2.798	7.509	4.734
Hviewer.exe	20	29	34	30

© 中国科学院软件研究所 http://www.jos.org.cn

图 13 更直观地给出了保护前后文件大小变化及 KeyCode 中的一条指令平均执行时间的变化.

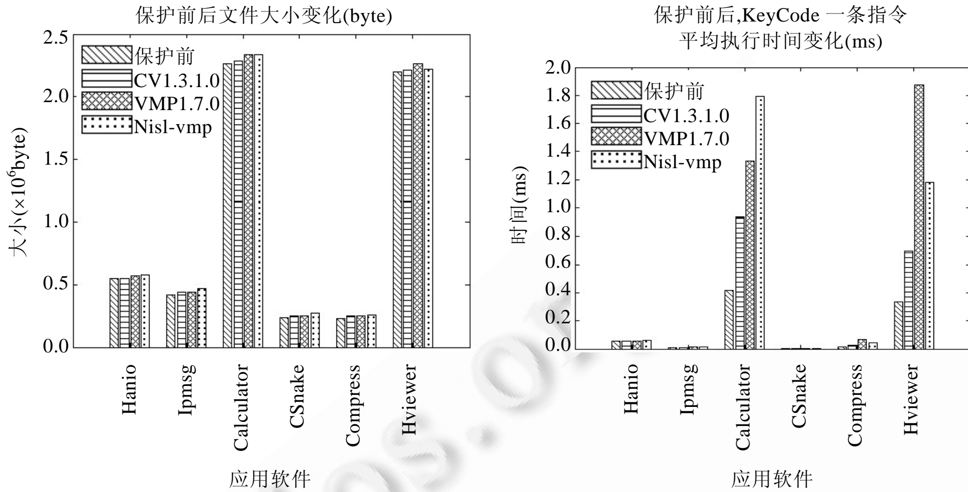

Fig.13 Performance of Nisl-vmp

图 13 Nisl-vmp 性能测试结果

从图 13 可以看出:

- 与商用虚拟机保护软件相比,Nisl-vmp 未增加空间开销;
- 但在一条指令的平均执行时间上,不同的实例,Nisl-vmp 的时间开销差别较大.

分析如下:

① 对于不同类型及同一类型不同寻址方式的 X86 指令,Nisl-vmp 解释执行时用到的 handler 序列不同,导致执行时间不同;

② 对于部分无法用已有的 handler 序列解释的 X86 指令,Nisl-vmp 需要暂时退出虚拟机,由真实主机执行后再进入虚拟机,这样会引起巨大的时间开销;

③ Nisl-vmp 的时间多样性效果使得执行不同执行路径的时间也不相同.

因此,针对于不同的保护实例,执行时间可能很小,也可能很大,需要进一步统计分析不同类型及寻址方式下的 X86 指令用不同 handler 序列解释时的执行时间等.这也是本文下一步需要继续深入研究的工作.

6 总结与展望

本文提出了一种具有时间多样性的虚拟机软件保护方法 TDVMP,解决了虚拟机软件保护技术抗累积攻击差的问题,从而达到提供更加持久保护的目的.通过实例实验表明:我们的方法具有较好的时间多样性效果,并且其性能开销在可接受范围.

总体上,本文提出的方法用于抗累积攻击很有意义,但由于目前用我们的方法保护后的软件在执行时采用随机方式选取执行路径,即,执行路径并不可控,这在实际应用中有时可能并非高效.比如,当保护后软件被运行于攻击模拟器中时,更理想的情况为有针对性地选择更抗模拟攻击的执行路径.

因此,如何使得被保护软件具有依据白盒攻击环境中存在的不同攻击威胁选取适当的执行路径,将是本文未来研究工作的一个重点.

致谢 在此,我们向对本文的工作给予支持和建议的学者,尤其是对开发 Nisl-vmp 做出贡献者张博、刘志伟、姜河表示感谢.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Chow S, Eisen P, Johnson H, van Oorschot PC. A white-box DES implementation for DRM applications. In: Feigenbaum J, ed. Proc. of the Digital Rights Management Workshop. LNCS 2696, Washington, 2002. 1−15. [doi: 10.1007/978-3-540-44993-5_1]

[2] Collberg C. The case for dynamic digital asset protection techniques [Ph.D. Thesis]. Department of Computer Science, University of Arizona, 2011. 1−5.

[3] Collberg C, Davidson JW, Giacobazzi R, Gu YX, Herzberg A, Wang FY. Toward digital asset protection. Intelligent Systems, 2011,26(6):8−13. [doi: 10.1109/MIS.2011.106]

[4] Anckaert B, de Bosschere K. Diversity for software protection [Ph.D. Thesis]. Ghent University, 2008.

[5] O'donnell AJ, Sethu H. Software diversity as a defense against viral propagation: Models and simulations. In: Proc. of the Symp. on Measurement, Modeling and Simulation of Malware. Monterey: IEEE, 2005. 247−253. [doi: 10.1109/PADS.2005.31]

[6] Bhatkar S. Defeating memory error exploits using automated software diversity [Ph.D. Thesis]. Stony Brook University in New York, 2007.

[7] Yang Y, Zhu S, Cao G. Improving sensor network immunity under worm attacks: A software diversity approach. In: Proc. of the 9th ACM Int'l Symp. on Mobile Ad Hoc Networking and Computing. New York: ACM Press, 2008. 149−158. [doi: 10.1145/1374618.1374640]

[8] De Sutter B, Anckaert B, Geiregat J, Chanet D, de Bosschere K. Instruction set limitation in support of software diversity. In: Lee PJ, Cheon JH, eds. Proc. of the Information Security and Cryptology (ICISC 2008). Seoul: Springer-Verlag, 2009. 152−165. [doi: 10.1007/978-3-642- 00730-9_10]

[9] Moser A, Kruegel C, Kirda E. Exploring multiple execution paths for malware analysis. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: ACM Press, 2007. 231−245. [doi: 10.1109/SP.2007.17]

[10] Technology O. Codevirtualizer. 2009. http://oreans.com/codevirtualizer.php

[11] Technologies O. Themida. 2013. http://oreans.com/themida.php

[12] Vmprotect. 2012. http://vmpsoft.com/products/vmprotect/

[13] Yang Z. Research and application of software diversity based on virtual machine [MS. Thesis]. Northwest University in Xi'an China, 2011 (in Chinese with English abstract).

[14] Udupa SKDSK, Madou M. Deobfuscation: Reverse engineering obfuscated code. In: Proc. of the 12th Working Conf. on Reverse Engineering (WCRE 2005). Pittsburgh: IEEE, 2005. 45−54. [doi: 10.1109/WCRE.2005.13]

[15] Williams D, Hu W, Davidson JW, Hiser JD, Knight JC, Tuong AN. Security through diversity: Leveraging virtual machine technology. IEEE Security and Privacy, 2009,7(1):26−33. [doi: 10.1109/MSP.2009.18]

[16] Collberg CMS, Myers J, Et Al. Remote tamper detection. 2012. http://digitalassetprotectionassociation.org/sites/default/files/RemoteTamperDetection.pdf

[17] Kil C, Jun J, Bookholt C, Xu J, Ning P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Proc. of the 22nd Annual of Computer Security Applications Conf. (ACSAC 2006). Shanghai: IEEE, 2006. 339−348. [doi: 10.1109/ACSAC.2006.9]

[18] Kc GS, Keromytis AD, Prevelakis V. Countering code-injection attacks with instruction-set randomization. In: Proc. of the 10th ACM Conf. on Computer and Communications Security. Washington: ACM Press, 2003. 272−280. [doi: 10.1145/948109.948146]

[19] Yang M, Huang LS. Software protection scheme via nested virtual machine. Journal of Chinese Computer Systems, 2011,32(2): 237−241 (in Chinese with English abstract).

[20] Jiao YK. Research and design on software protection system based on virtual machine [MS. Thesis]. Wuhan: Huazhong University of Science and Technology, 2007 (in Chinese with English abstract).

[21] Ghosh S, Hiser J, Davidson JW. Replacement attacks against VM-protected applications. In: Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments. London: ACM Press, 2012. 203−214. [doi: 10.1145/2151024.2151051]

[22] Desai P. Towards an undetectable computer virus [Ph.D. Thesis]. San Jose State University, 2008.

© 中国科学院软件研究所 http://www.jos.org.cn

附中文参考文献:

[13] 杨朕.基于虚拟机的软件多样性研究与应用[硕士学位论文].西安:西北大学,2011.

[19] 杨明,黄刘生.一种采用嵌套虚拟机的软件保护方案.小型微型计算机系统,2011,32(2):237–241.

[20] 焦义奎.基于虚拟机的软件保护系统研究与设计[硕士学位论文].武汉:华中科技大学,2007.

房鼎益(1959－),男,陕西汉中人,博士,教授,博士生导师,CCF 高级会员,主要研究领域为网络与信息安全,软件安全与保护,无线传感器网络关键技术及其研究应用.

顾元祥(1951－),男,教授,首席架构师,主要研究领域为计算机系统安全与保护,软件安全与保护,数字内容安全与保护,白箱安全与可信性.

赵媛(1989－),女,硕士生,主要研究领域为软件安全与保护,软件攻击技术.

许广莲(1990－),女,硕士生,主要研究领域为软件安全与保护,软件攻击技术.

王怀军(1981－),男,博士生,讲师,CCF 会员,主要研究领域为软件安全与保护,软件攻击及软件保护有效性评测.

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn