基于指针时空分析的软件异常可利用性判定*

彭建山，王清贤，欧阳永基
（解放军信息工程大学 数学工程与先进计算国家重点实验室，郑州 450002）

摘 要： 软件异常的可利用性是评估漏洞威胁等级的重要指标。针对目前二进制程序异常可利用性判定方法存在检测模式少、分析深度和精度不足、准确率低的问题，通过分析多重指针的时空局部有效性，利用独立可控数据的内存布局构造从异常指令到跳转指令的指针引用路径，能够快速收敛搜索域，提高控制流劫持成功率和判定准确率。实验结果表明，该方法适用于栈溢出、堆溢出、整型溢出导致覆盖返回地址、函数指针等模式，具有较高的准确率和较小的性能开销。

关键词： 可利用性判定；污点分析；控制流劫持；多重指针引用
中图分类号： TP311.53 **文献标志码：** A **文章编号：** 1001-3695(2016)05-1504-05
doi:10.3969/j.issn.1001-3695.2016.05.050

Exploitable inference based on space-time analysis of pointers

Peng Jianshan, Wang Qingxian, Ouyang Yongji
(*State Key Laboratory of Mathematics Engineering & Advanced Computing, PLA Information Engineering University, Zhengzhou* 450002, *China*)

Abstract: Whether exception can be exploitable is important to evaluate threaten level of vulnerability. There are some problems in the current automation exploitable inference methods for binary program: lack of exploiting model, or less depth and precision in analysis. These problems result in a high rate of false positives and false negatives. To reduce searching range in memory and improve the exploitable rate, this paper researched local effectiveness of space-time of multiple pointers, used the independent controllable data to construct the path from the exception instruction to a jump instruction. The test results show that, this method is applicable to the stack overflow, heap overflow and integer overflow caused by the return address and function pointer rewriting. Its accuracy and time performance are better than similar tools.

Key words: exploitable inference; taint analysis; control-flow hijack; multiple pointers dereference

0 引言

目前已有大量的软件安全测试技术用于发现二进制程序中存在的异常(exception)，如模糊测试、动态符号执行[1]等。这些技术大多以发现异常为测试终止条件，而没有或无法对异常的可利用性(exploitable)作进一步分析和判定，即缺少自动化评估程序异常的威胁严重等级的环节。

在早期软件安全测试过程中，判定软件异常的可利用性通常由分析人员借助调试器、Valgrind[2]和 AddressSanitizer[3]等工具完成，严重依赖分析人员的经验和能力，也存在分析精度和深度不足问题，误判和漏判率较高。针对上述问题，研究人员提出了多种解决方法，APEG[4]是基于补丁(patch)分析的漏洞利用攻击代码自动生成工具，但需要程序补丁的支持。AEG[5]借鉴了 APEG 关于漏洞利用过程可被描述为程序状态空间谓词的思想，能够自动完成从异常发现、可利用性分析到攻击代码生成的全过程，但该工具针对源代码，只能处理栈返回地址覆盖和格式化字符串溢出两种漏洞类型。MAYHEM[6]结合了在线式和离线式符号执行的优点，主要用于二进制程序的测试，其中的异常可利用性判定功能基于 AEG 改造实现，但由于缺乏高级语言的类型和结构信息的支持，判定的效率和准确度反而较 AEG 更差。此外，Microsoft 公司开发了基于 Windbg 调试器的! Exploitable 工具[7]，当程序在调试过程中发生 crash 时可辅助判断其可利用性，该工具静态分析 crash 发生时的栈环境和代码上下文环境，缺乏对后续数据流和控制流的动态分析，准确率非常低。

本文分析软件异常可利用性判定所面临的问题，研究利用过程中控制流和数据流的构造规律，分析多重指针引用的时空局部有效性和依赖关系，提出了针对二进制程序的基于指针时空分析的判定方法，旨在解决检测模式少、准确率低等问题。

1 问题和难点分析

当程序发生异常时，执行过程被中断，程序的后继状态变得难以预测，而异常可利用性判定的是后继状态中是否存在劫持控制流的可能。

先给出与可利用性相关的两个定义：

定义 1 如果一个寄存器，或内存地址，或内存地址指向的内容与用户输入直接相关，即可以由用户的输入完全控制，则称该寄存器或内存地址或内存内容是可控的。

这里可控数据与污点数据[8](taint)并不是一个概念，污点数据指的是数据被用户输入影响，既可能控制多个字节，也可

收稿日期：2015-01-31；修回日期：2015-03-07 基金项目：国家“863”计划资助项目(2012AA012902)
作者简介：彭建山(1979-)，男，江西吉安人，讲师，博士研究生，主要研究方向为网络安全、软件安全(jxpjs@163.com)；王清贤(1960-)，男，教授，硕士，主要研究方向为网络安全；欧阳永基(1985-)，男，博士研究生，主要研究方向为网络安全、软件安全.

能只控制几个比特,如:

$$\text{taint: u8} = \text{input: u8} << 4$$

污点数据 taint 只有高 4 位能够被用户输入 Input 所控制。而可控数据指的是通过用户输入可以构造出任意的数据,显然可控数据是污点数据的一个子集。

定义 2　程序异常是可利用的,当且仅当该异常指令之后存在跳转指令 Jump addr/Jump [addr],且满足如下条件之一: 跳转目标地址 addr 是可控的,或跳转地址指向的内容 [addr] 是可控的。

这里跳转指令包含了 call、ret、jmp 等指令。以上定义不考虑保护机制的影响,如 ASLR、DEP 等。

以图 1 代码为例,Exploitable 工具不能准确判定该代码的可利用性,因为它的判定依据如下:

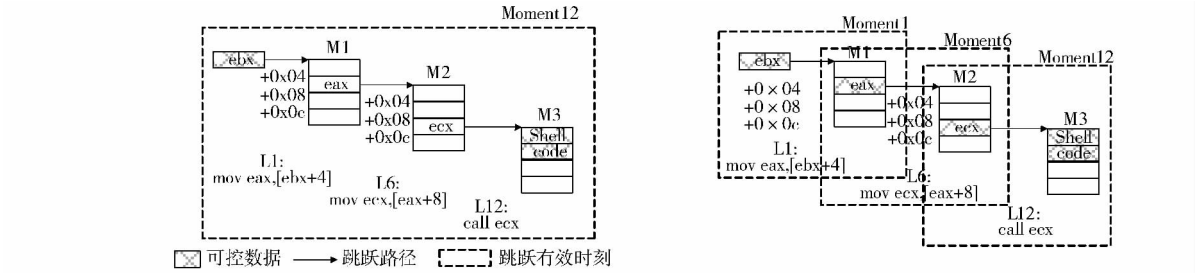
a) 发生异常指令的类型。如果是读内存指令,判定为不可利用; 如果是写内存指令,判定为可能可以利用。L1 为读内存指令,会错误地判定为不可利用。

b) 异常指令的后续若干条指令中是否存在跳转指令。如果 L12 相隔 L1 较远,则判定不可利用。

```
L1      mov eax, dword ptr [ebx + 4]
  //ebx = 0x41414141,触发内存访问异常,且 ebx 可控
L2-L5   …
L6      mov ecx, dword ptr [eax + 8]
L7-L11  …
L12     call ecx
```

图 1　异常示例代码

因此仅静态分析局部代码是不够的,需要对程序的后续控制流和数据流作精细化分析。很显然,因为 L1 的 ebx 寄存器可控,如果能够找到一块内存地址 M1,且 M1 + 4 指向的内容为地址 M2,同时满足 M2 + 8 指向的内容为地址 M3,且 M3 可控或 M3 指向的内容可控,那么该异常就是可利用的。内存分布如图 2(a) 所示,需要找到图 2(a) 中的三块内存 M1、M2、M3,其中灰色标示的区域为可控数据。构造一条从头指针 ebx 逐级“跳跃”(从前一级指针指向下一级内存) 最终到达 M3 的路径,本文称之为跳跃路径。



(a) 反推法: 跳跃指针范围为全部内存,且必须在时刻Moment12 (L12指令) 时有效　(b) 时空分析法: 跳跃指针范围为可控数据、且只需在跳跃时有效

图 2　内存分布图

人工分析时,为了构造跳跃路径通常采用反推法: 从 L12 的 call ecx 指令开始,检查 ecx 是否可控,如果不可控,获得可控数据集合 $S_C = \{T_1, T_2, \cdots, T_n\}$; 回溯至 L6,在当前内存中搜索满足如下约束的解 eax:

$$\{0 \leqslant \text{eax} < 2^{32} \wedge [\text{eax} + 8] == T_i \mid T_i \in S_C\}$$

如果无解,说明异常不可利用,否则得到解集合 S_1。再回溯至 L1,搜索当前内存中满足与 L1 相关约束的解 ebx:

$$\{0 \leqslant \text{ebx} < 2^{32} \wedge [\text{ebx} + 4] == \text{eax} \mid \text{eax} \in S_1\}$$

依此类推最终得到一条跳跃路径。反推法存在两个难以实现自动化的问题:

a) 每次回溯都要对全部内存进行搜索,搜索次数为可控数据大小 × 内存大小 × 回溯次数,搜索域太大,搜索时间过长。

b) 在最终执行跳转指令(L5) 时,要保证跳跃路径上的每级节点都要全局有效,而在实际情况中找到一条这样的路径十分困难,跳跃级数越多,可能性就越小。

针对以上问题,本文提出了一种基于指针时空分析的判定方法,使程序能够在不同的执行时刻依照当时的可控数据分布情况,构造一条跳跃路径,如图 2(b) 所示。与图 2(a) 的区别在于:

a) 图中的阴影位置不同: 每一级指针不是在全部内存中搜索,而是在可控数据中选择,计算可控数据构造多重指针引用(空间分析) ;

b) 每次跳跃的有效时刻(moment) 范围不同: 不需要每次跳跃的环境都全局有效,而只需要在跳跃的当前时刻有效(时间分析) 。

该方法是一种动态的控制流和数据流精细化分析方法,不仅极大缩小了搜索域,减少了搜索时间,同时由于每次跳跃只依赖于局部时间的指针引用关系,提高了构造跳跃路径即控制流劫持路径的可能性。

实现该方法的难点在于如何建立可控数据间的多重指针引用关系。如图 2(b) 所示,需要精确计算 ebx、M1 中的 eax 和 M2 中的 ecx 的值,保证控制流能够执行到 L12 行,而且跳转到可控数据。

2　基于多重指针时空分析的可利用性判定

为解决以上难点,本文以动态污点分析[8]为基础,通过分析独立可控数据分布,保持控制流,构建引用关系集合并求解指针,构造跳跃路径,最后生成样本实际验证判定的准确性。动态污点分析是程序分析的重要方法,通过将用户输入标记为污点数据,在程序执行过程中分析污点数据的传播过程(propagation) ,可得到任意时刻污点数据在内存和寄存器中的分布情况。

2.1　分析独立可控数据的分布

由于可利用性判定只关注可控数据,需要对污点分析加入新的约束:

a) 位运算去除污点标记。

b) 非线性运算如乘、除、模运算等,去除污点标记。

c) 污点分析有两种策略[9]，即 overtainting 和 undertainting，为避免污点传播范围过大，保证可控数据的有效性，缩小搜索域，本文采用 undertainting 策略。

d) 由于本文关注的数据主要是内存地址，所以污点分析的粒度达到字节级即可，而不需要虽然精度更高但效率更低的比特级。

以上约束会导致牺牲一些污点分析的精度，但从实际应用效果来看，这种牺牲是可接受的。

如果可控数据间存在传播关系，例如以下指令会将 T_1 传播给 T_2：

mov eax，T_1

mov T_2，eax

T_1 和 T_2 其实指向同一个污点来源，可控数据的非独立性会在构造指向关系时产生矛盾，在分析时需要将它们视做同一数据。

通过跟踪分析独立可控数据的传播，可以得到程序在任意时刻独立可控数据的分布情况，包括内存分布和寄存器分布。

2.2 执行步骤

当程序发生异常后，后续的执行流程不再是预定的流程，而判定可利用性必须要分析预定流程（覆盖异常处理结构链时例外，在本文第 3 章中讨论）。例如在图 1 中，执行 L1 代码会导致程序异常，L2 ~ L12 的代码将不会被执行；另一个可能的异常点是 L6，则 L7 ~ L12 的代码将不会被执行，而控制流劫持发生在 L12。因此在判定时需要使程序不发生异常，保持预定流程。

本文使用 Pin[10] 作为基础平台，既用于动态污点分析，也用于控制流和数据流处理。为保持程序的预定流程，采用以下步骤：

a) 在指令执行前暂停线程，调用 IsMemoryOperation 函数，检查是否内存操作指令，是则继续，否则恢复线程，重复步骤 a)。

b) 调用 IsControlledAddress 函数，检查内存地址是否是可控数据，是则继续，否则恢复线程，重复步骤 a)。

c) 调用 PredictAddress 函数，获得一个有效的内存地址（见本文 2.3 节）。

d) 用新地址代替指令中的地址，调用 IsJumpOperation 函数，检查是否跳转指令，是则继续，否则恢复线程，重复步骤 a)。

e) 判定异常可利用性（见本文 2.4 节），是则终止，否则恢复线程，回到步骤 a)。

仍以图 1 为例，执行过程如图 3 所示。

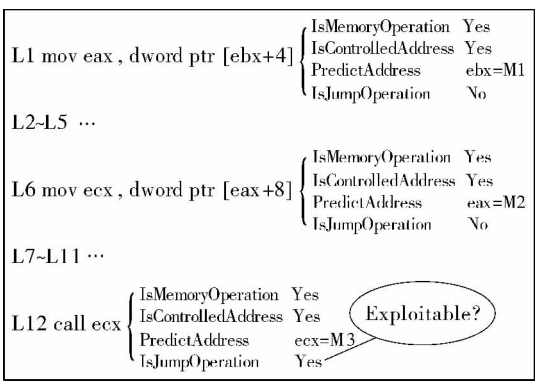


图 3 Pin 插桩处理过程

L12 行的 call ecx 指令也被认做内存操作指令，因为它将访问 ecx 所指向的内存，将其内容作为下一条指令。

2.3 引用关系集合的构建和指针求解

反推法计算指针的问题在于每次寻找可用指针时都需要搜索整个内存，搜索域过大导致效率低下，且搜索到有效指针的成功率很低。本文提出一种基于引用关系集合和指针求解的正向搜索方法，能够快速收敛搜索域，在可接受的时间内完成跳跃路径的构造，而且保证了较高的成功率。

该方法的基本思想是：从异常指令开始，遇到 PredictAddress 函数时，如 L1 行的第一次读内存操作，分析此刻的独立可控数据的内存分布，假设为集合

$$S_c = \{ [P_1] = T_1, [P_2] = T_2, \cdots, [P_n] = T_n \}$$

其中：T 为 4Byte 的可控数据，P 为 T 所在内存的地址。按照 L1 指令的语义，构建第一层引用关系集合：

$$\lambda_1 = \begin{cases} (P_1 + 4) \rightarrow T_1 \\ \cdots \\ (P_n + 4) \rightarrow T_n \end{cases}, T_i \in S_C, 1 \leqslant i \leqslant n$$

其中：表达式 $x \rightarrow y$ 表示 $x = \&y$。注意到此时关心的是 T_i 的地址，其具体值还没有确定。

PredictAddress 函数随机选择 $\{P_i \mid 1 \leqslant i \leqslant n\}$ 作为返回值，保证程序不发生异常。

当执行到 L6 行的第二次读内存操作时，首先更新 S_C，进而更新 λ_1，尤其是删除包含已消除可控数据的元素。由于后续的引用只与 λ_1 中的可控数据 T_i 有关，所以将 λ_1 中的 T_i 分为数量相等的两部分，一部分用于指针，另一部分用于指针指向的内容。构建第二层引用关系集合如下：

$$\lambda_2 = \begin{cases} (T_1 + 8) \rightarrow T'_1 \\ \cdots \\ (T_p + 8) \rightarrow T'_p \end{cases}$$

$$T_i, T'_j \in S_C, T_i \neq T'_j, 1 \leqslant i \leqslant p, 1 \leqslant j \leqslant p$$

注意到 λ_1 中的 T_i 和 λ_2 中的 T_i 是对应的。为确定 T_i 的具体值，以 P_i 和 $\&T'_i$ 为常量，T_i 为变量进行求解，获得 T_i 的可行解代入 λ_2，PredictAddress 函数随机选择一个解作为返回值。

依此类推，执行到 L12 行的第三次读内存操作时，更新 S_C 和 λ_2，构建第三层引用关系集合如下：

$$\lambda_3 = \begin{cases} T'_1 \rightarrow T''_1 \\ \cdots \\ T'_q \rightarrow T''_q \end{cases}$$

$$T'_j, T''_k \in S_C, T'_j \neq T''_k, 1 \leqslant j \leqslant q, 1 \leqslant k \leqslant q, q \leqslant \frac{p}{2}$$

以 T_i 和 T''_i 为常量，以 T'_i 为变量进行求解，获得 T'_i 的可行解代入 λ_3，PredictAddress 函数随机选择一个解作为返回值。

图 4 示意了构建集合的过程。与反推法相比，在构建第一层集合时，搜索域是集合 S_C，远小于全部内存。而以后每构建一层，搜索域都会减半，搜索域的快速收敛极大降低了全部搜索耗费的时间。同时可以看到，每次 PredictAddress 函数返回的指针在当前时刻是有效的，这就保证了跳跃路径中的每级指针在当前时刻的有效性。

容易证明: 设跳跃次数为 n,如果$\{|\lambda_i|\} \geq 2^{n-i}, 1 \leq i < n\}$,则 $\lambda_n \neq \varnothing$,此时可以构造出至少一条如图 2(b) 所示的跳跃路径,保证了较高的成功率。

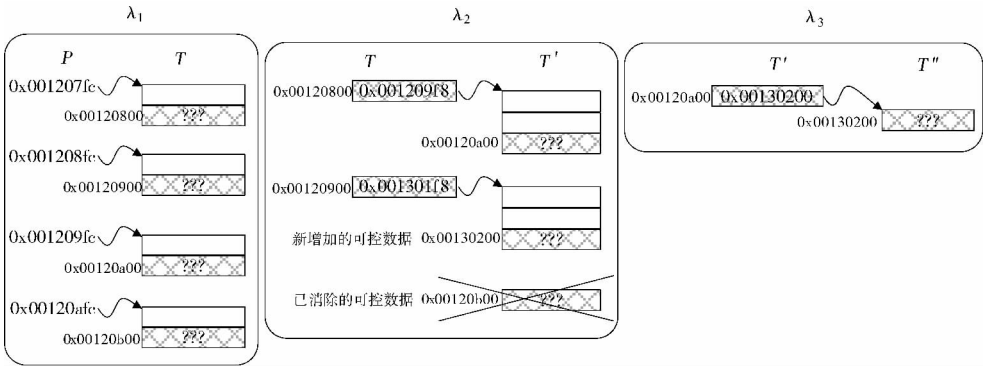


图 4　引用关系集合构建和指针求解示意图

为保证集合的准确性和实时性,在可控数据的内存分布发生变化(S_C 集合发生变化) ,如产生新数据,或消除旧数据时,都要及时更新上一层集合,这会带来一些性能上的损失,但每次更新量都很小,而且这种更新可以看做是动态污点分析的一部分。为了优化性能,本文使用惰性更新法,即在每次构建新的层次时再更新所有的集合。

2.4　可利用性判定

2.3 节的方法没有考虑以下两个问题:

a) 地址不稳定,如程序中同一次堆的创建在每次运行时都可能会不同。

b) 某时刻新产生的可控数据可能与之前时刻已消除的可控数据指向同一污点来源,这和非独立可控数据一样在构造引用关系时会产生矛盾。

因此为了保证判定的准确性,需要对每个可行解生成样本,并通过实际运行验证其有效性。生成样本从第一层集合开始,逐级取得集合中级联的指针,得到若干条跳跃路径。通过动态污点分析建立的污点与样本关系,将每条路径的实际值代入旧样本生成若干个新样本,最后以新样本作为输入运行程序,验证可利用性,如算法 1 所示。

算法 1

```
Input: crash sample, {λ1, λ2, ⋯, λn}.
Output: Exploitable or not exploitable.
foreach (Pointer→ControlledData) ∈ λ1 do
  for i = 2 to n
    foreach Pointer´ in λi do
    if Pointer´ == ControlledData then
        Pointeri = Pointer;
        Pointer = ControlledData in λi
        break;
      end if
    end for
    new sample = generatenewsample(Pointer1, Pointer2, ⋯, Pointern, crash sample);
    testsample(new sample);
    if jump into controlled data then
      output " Current sample is exploitable"
    else
      output "Current sample is not exploitable"
    end if
  end
```

这里的 $\lambda_1, \lambda_2, \cdots, \lambda_n$ 指的是其构建时的内容,而不是之后更新后的内容。

3　实验与分析

根据以上分析方法,基于 Pin 平台实现了一个软件异常可利用性判定工具 BEE(binary exception exploitable) ,包括动态污点分析、控制流分析、引用关系集合构建和指针求解、样本生成和可利用性判定等模块。

实验对象为存在已知漏洞的暴风影音、Word 等应用程序,操作系统为 Windows XP SP3 中文版升级最新补丁,主机配置为 Intel i7 3.6 GHz 处理器,8 GB 内存,1 TB 硬盘。实验过程: 先使用 fuzz 工具对测试程序进行测试,对 crash 样本用 BEE 工具判定异常的可利用性。表 1 是测试结果。

实验结果分析:

a) BEE 工具能够对多种类型的软件异常进行可利用性判定,准确率远高于 Exploitable 工具; 因为 MAYHEM 采用了更为复杂的符号执行技术,所以 BEE 的分析耗时远小于 MAYHEM。

b) 2 号测试程序的误判原因分析: 该漏洞需要通过覆盖 SEH 来利用。由于 BEE 工具改变了每个异常指令的操作数,指令不再出错,程序不会进入异常处理过程,所以这里产生了误判(1 号程序没有误判是因为栈溢出时还覆盖了返回地址) 。

c) 构建引用关系集合时使用了简单的二分法划分指针和内容,会造成可控数据的浪费。当可控数据很少时,需要采用更为精细的分析。

d) BEE 工具测试的是单路径,对于多路径(即异常指令的后续路径分支条件受污点数据影响) 情况,只能选择一条路径处理。

e) BEE 工具没有考虑 ASLR 机制[11] 的影响,在 Windows 7 环境中可能会因为可控数据地址不稳定导致指针引用失败。

4　结束语

本文提出了一种基于指针时空分析的软件异常可利用性判定方法,能够避免在全局内存空间中搜索有效指针,缩短了

搜索时间;充分利用可控数据的局部有效性构造多重指针引用路径,提高了控制流劫持成功率和判定准确率。该方法适用于二进制程序的多种异常类型,但是存在两个主要问题: a) 只能分析异常指令后的单路径,分析多路径可能需要结合动态符号执行技术; b) ASLR 等机制会影响判定的准确性,需要进一步研究解决方法。

表 1 BEE 工具测试结果

序号	被测程序	漏洞编号	漏洞成因	crash 原因	样本大小	分析耗时/s	测试结果
1	Soritong 1.0	CVE-2009-1643	栈溢出 覆盖 SEH 和返回地址	ret 指令出错	1000 Byte	25 845 **	可以利用 可以利用*
2	暴风影音 2012 3.10.4.8	CNVD-2010-00752	栈溢出 覆盖 SEH	写内存指令出错	1 KB	12	不可利用 可能可以利用*
3	Word 2003 11.0.8324.0	CVE-2010-3333	栈溢出 覆盖返回地址	读内存指令出错	12 KB	126	可以利用 不可利用*
4	Word 2003 11.0.8331.0	CVE-2012-0177	堆溢出 覆盖函数指针	读内存指令出错	15 KB	130	可以利用 不可利用*
5	Word 2007 12.0.4518.1014	CVE-2013-3906	堆溢出 + 整型溢出 覆盖函数指针	读内存指令出错	20 KB	104	可以利用 不可利用*

注: * 标注 Exploitable 工具的测试结果, ** 标注 MAYHEM 工具的测试结果。

参考文献:

[1] Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing [C] //Proc of Network and Distributed System Security Symposium. 2008: 151-166.

[2] Nethercote N, Seward J. Valgrind: a program supervision framework [J]. Electronic Notes in Theoretical Computer Science, 2003, 89(2): 44-66.

[3] Serebryany K, Bruening D, Potapenko A, *et al*. AddressSanitizer: a fast address sanity checker [C] //Proc of USENIX Annual Technical Conference. 2012: 309-318.

[4] Brumley D, Poosankam P, Song D, *et al*. Automatic patch-based exploit generation is possible: techniques and implications [EB/OL]. (2008-05-01) [2015-01-30]. http://repository.cmu.edu/cgi/viewcontent.cgi? article = 1001&context = ece.

[5] Avgerinos T, Cha S K, Rebert A, *et al*. Automatic exploit generation [J]. Communications of the ACM, 2014, 57(2): 74-84.

[6] Cha S K, Avgerinos T, Rebert A, *et al*. Unleashing MAYHEM on binary code [C] //Proc of IEEE Symposium on Security and Privacy. [S.l.]: IEEE Press, 2012: 380-394.

[7] ! Exploitable crash analyzer-MSEC debugger extensions [EB/OL]. (2013-05-01) [2015-01-30]. http://msecdbg.codeplex.com/.

[8] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [C] //Proc of Network and Distributed System Security Symposium. 2005.

[9] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution [C] //Proc of IEEE Symposium on Security and Privacy. [S.l.]: IEEE Press, 2010: 317-331.

[10] Pin: a dynamic binary instrumentation tool [EB/OL]. (2012-06-13) [2015-01-30]. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[11] Howard M, Miller M, Lambert J, *et al*. Windows ISV software security defenses [EB/OL]. (2010-12-01) [2015-01-30]. https://msdn.microsoft.com/en-us/library/bb43 0720.aspx.

(上接第 1490 页)

[3] Zhang Jin, Zhang Qian. Stackelberg game for utility-based cooperative cognitive radio networks [C] //Proc of the 10th ACM International Symposium on Mobile Ad hoc Networking and Computing. New York: ACM Press, 2009: 23-32.

[4] Wang Xiaocheng, Guan Xinping, Ma Kai, *et al*. Pricing-based spectrum leasing in cognitive radio networks [C] //Proc of Chinese Control Conference. 2012: 5476-5480.

[5] Naeem M, Lee D C, Pareek U. An efficient multiple relay selection scheme for cognitive radio systems [C] //Proc of IEEE International Conference on Communications. 2010: 1-5.

[6] Nam S, Vu M, Tarokh V. Relay selection methods for wireless cooperative communications [C] //Proc of the 42nd Annual Conference on Information Sciences and Systems. 2008: 859-864.

[7] Jing Yindi, Jafarkhani H. Single and multiple relay selection schemes and their achievable diversity orders [J]. IEEE Trans on Wireless Communications, 2009, 8(3): 1414-1423.

[8] Luo Ronghua, Yang Zhen. Stackelberg game-based distributed power allocation algorithm in cognitive radios [J]. Journal of Electronics & Information Technology, 2010, 32(12): 2964-2969.

[9] Luo Gaofeng, Wei Renyong. Auction-based spectrum allocation in cognitive radio system [J]. Communications Technology, 2010, 43(9): 48-49, 53.

[10] Bletsas A, Khisti A, Reed D P. A simple cooperative diversity method based on network path selection [J]. IEEE Journal on Selected Areas in Communications, 2006, 24(3): 659-672.

[11] Wang Beibei, Han Zhu, Liu K J R. Distributed relay selection and power control for multiuser cooperative communication networks using Stackelberg game [J]. IEEE Trans on Mobile Computing, 2009, 8(7): 975-990.

[12] 张琰, 盛敏, 李建东. 协作通信网络中的中继节点选择技术 [J]. 中兴通信技术, 2010, 16(1): 23-27.