计算机科学
COMPUTER SCIENCE

基于深度学习的Linux系统DKOM攻击检测

陈亮, 孙聪

引用本文

陈亮, 孙聪. 基于深度学习的Linux系统DKOM攻击检测[J]. 计算机科学, 2024, 51(9): 383-392.

CHEN Liang, SUN Cong. Deep-learning Based DKOM Attack Detection for Linux System[J]. Computer Science, 2024, 51(9): 383-392.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230700035

基于深度学习的 Linux 系统 DKOM 攻击检测

陈　亮[1,2]　孙　聪[1]
1 西安电子科技大学网络与信息安全学院　西安 710071
2 华为技术有限公司　西安 710100
(18829056730@163.com)

摘　要　直接内核对象操纵(DKOM)攻击通过直接访问和修改内核对象来隐藏内核对象,是主流操作系统长期存在的关键安全问题。对 DKOM 攻击进行基于行为的在线扫描适用的恶意程序类型有限且检测过程本身易受 DKOM 攻击影响。近年来,针对潜在受 DKOM 攻击的系统进行基于内存取证的静态分析成为一种有效和安全的检测方法。现有方法已能够针对 Windows 内核对象采用图神经网络模型进行内核对象识别,但不适用于 Linux 系统内核对象,且对于缺少指针字段的小内核对象的识别有效性有限。针对以上问题,设计并实现了一种基于深度学习的 Linux 系统 DKOM 攻击检测方案。首先提出了一种扩展内存图结构刻画内核对象的指针指向关系和常量字段特征,利用关系图卷积网络对扩展内存图的拓扑结构进行学习以实现内存图节点分类,使用基于投票的对象推测算法得出内核对象地址,并通过与现有分析框架 Volatility 的识别结果对比实现对 Linux 系统 DKOM 攻击的检测。提出的扩展内存图结构相比现有的内存图结构能更好地表示缺乏指针但具有常量字段的小内核数据结构的特征,实现更高的内核对象检测有效性。与现有基于行为的在线扫描工具 chkrootkit 相比,针对 5 种现实世界 Rootkit 的 DKOM 行为,所提方案实现了更高的检测有效性,精确度提高 20.1%,召回率提高 32.4%。

关键词:内存取证;恶意软件检测;操作系统安全;图神经网络;二进制分析
中图分类号　TP309

Deep-learning Based DKOM Attack Detection for Linux System

CHEN Liang[1,2] and SUN Cong[1]
1 School of Cyber Engineering,Xidian University,Xi'an 710071,China
2 Huawei Technologies Co.,Ltd.,Xi'an 710100,China

Abstract　Direct kernel object manipulation(DKOM) attacks hide the kernel objects through direct access and modification to the kernel objects. Such attacks are a long-term critical security issue in mainstream operating systems. The behavior-based online scanning can efficiently detect limited types of DKOM attacks,and the detection procedure can be easily affected by the attacks. In recent years,memory-forensics-based static analysis has become an effective and secure detection approach in the systems potentially attacked by DKOM. The state-of-the-art approach can identify the Windows system kernel objects using a graph neural network model. However,this approach cannot be adapted to Linux kernel objects and has limitations in identifying small kernel objects with few pointer fields. This paper designs and implements a deep-learning-based DKOM attack detection approach for Linux systems to address these issues. An extended memory graph structure is proposed to depict the points-to relation and the constant fields' characteristics of the kernel objects. This paper uses relational graph convolutional networks to learn the topology of the extended memory graph to classify the graph nodes. A voting-based object inference algorithm is proposed to identify the kernel objects' addresses. The DKOM attack is detected by comparing our kernel object identification results and the results of the memory forensics framework Volatility. The contributions of this paper are as follows. 1) An extended memory graph structure that improves the detection effectiveness of the existing memory graph on capturing the features of small kernel data structures with few pointers but with evident constant fields. 2) On the DKOM attacks raised by five real-world Rootkits,our approach achieves 20.1% higher precision and 32.4% higher recall than the existing behavior-based online scanning tool chkrootkit.

Keywords　Memory forensics,Malware detection,Operating system security,Graph neural network,Binary analysis

到稿日期:2023-07-06　返修日期:2023-11-14
基金项目:国家自然科学基金(62272366);陕西省重点研发计划(2023-YBGY-371)
This work was supported by the National Natural Science Foundation of China(62272366) and Key Research and Development Program of Shaanxi Province(2023-YBGY-371).
通信作者:孙聪(suncong@xidian.edu.cn)

1 引言

内核级恶意软件运行于内核空间，能够获取更高的系统权限，因而相较于一般恶意软件具有更强的隐蔽性与危害性。Rootkit 是通过修改关键系统数据结构来隐藏自身并为木马提供后门的恶意程序。内核级 Rootkit 是一类重要的内核级恶意软件，通过内核钩子、内核补丁或直接内核对象操纵(Direct Kernel Object Manipulation，DKOM)技术长期潜伏在系统中，盗取数据、劫持控制流或配合其他恶意程序进一步实施攻击[1]。基于内核钩子和内核补丁的内核级 Rootkit 均需修改系统调用的相关入口和函数，在内核完整性检查中容易暴露；DKOM 技术则直接修改恶意程序在系统内核中的关键数据结构，隐蔽性较高。具体地，内核级 Rootkit 利用 DKOM 技术对系统进程、端口、内核模块等内核动态对象进行指定修改，由于这些核心内核对象是系统调用查询信息的数据源，因而修改能够隐藏信息源并绕过完整性检查[2]。例如，将属于恶意程序的内核进程对象从内核对象列表中分离，会导致系统在扫描内核进程时无法发现恶意进程。有效地检测出基于 DKOM 技术的内核 Rootkit 恶意攻击(简称 DKOM 攻击)，对操作系统安全具有重要意义。

为了检测 DKOM 攻击，一些方法[3-4]根据事先记录的 DKOM 恶意程序的代码特征或者攻击行为特征，通过判断系统内核中运行的程序是否符合所记录的特征来判断系统是否受到攻击。这类方法的局限性在于只能检测到已知特征的 DKOM 恶意程序，而对于未掌握其特征的未知 DKOM 攻击则无法检测到。另一些方法[5-6]通过内核数据结构的不变量签名来检测系统内存镜像中的所有内核对象，并与内核列表中的内核对象进行对比来检测 DKOM 攻击，然而并非每种内核数据结构都拥有不变量签名。针对 DKOM 攻击通过修改内核对象指针隐藏自身的特点，我们可以对操作系统内存转储镜像进行内核数据查找，通过判断查找结果中是否包含被恶意隐藏的内核数据结构，来检测 DKOM 攻击。

本文提出了一种基于深度学习的 Linux 系统 DKOM 攻击检测方案。首先，对 Linux 系统内存镜像进行转储后的静态分析，基于内核数据结构中的指针映射与值不变字段构造了一种扩展内存图结构。然后利用关系图卷积神经网络对扩展内存图的拓扑结构进行学习以实现节点分类模型。在节点分类结果基础上，使用基于投票的对象推测算法得出内存中的所有内核对象地址。最后通过对比与内存分析框架 Volatility 识别的内核对象列表的差异，检查存在于内核对象列表外的内核对象，从而判断 DKOM 攻击的存在性。本方案不依赖内核数据结构的不变量签名，且充分利用了内核数据结构中的指针与常量字段特征，能够高精度地检测内存镜像中的内核对象及多种 DKOM 攻击。本文的主要贡献如下：

1)提出了适用于 Linux 内存镜像的扩展内存图结构，实现了完整的扩展内存图构造工具链。相比 DeepMem[7]的 Windows 内存图结构，本文提出的扩展内存图结构能够更好地表示缺乏指针但具有常量字段的内核数据结构的特征，对 module_attribute 结构的识别说明了针对此类内核数据结构的识别性能提升。

2)基于关系图卷积网络和从正常内核镜像中分析得出的节点标签，学习扩展内存图节点特征并实现分类器，进而实现对 Linux 内核 task_struct 等 7 类关键内核数据结构的节点分类和数据结构识别。

3)本文实现的 Linux 系统 DKOM 攻击检测工具与现有基于行为的检测工具 chkrootkit 的实验结果对比显示，对于 adore-ng，Wukong，KBeast，Jynx2，LilyOfTheValley 这 5 种现实世界内核级 Rootkit 的 DKOM 攻击，相比 chkrootkit，本文工具检测精确度提高 20.1%，召回率提高 32.4%。

2 相关工作

现有的 Linux 系统 DKOM 攻击检测方案主要分为分析系统内核数据结构和分析恶意软件行为两类。

DKOM 攻击通过直接修改内核数据结构的方式实现隐藏进程、隐藏文件等目的，导致操作系统能够识别的各种内核数据结构(如进程队列等)与实际内存中的内核对象不一致。因此，从操作系统内存镜像中识别内核数据结构是一项重要的基础性工作。根据内存搜索技术的不同，具体方法可分为两类：基于对象列表遍历的方法和基于签名扫描的方法。

基于对象列表遍历的技术通过遍历操作系统的内核链表来查找所有内核对象[8-9]，Linux 系统会在内核中维护一个链表用来管理内核数据结构，如所有 task_struct 都存储在一个全局链表中，每个 task_struct 都有一个 prev 指针和一个 next 指针，分别指向前驱和后继 task_struct，task_struct 链表的首节点是全局变量 init_task，列表遍历方法通常从该首节点开始遍历整个链表来获得所有相关对象。在链表遍历过程中可能遇到泛型指针及动态数组等模糊数据结构，KOP[8]运用过程间指向分析来计算泛型指针的所有可能类型，使用模式匹配算法来解决类型歧义，并利用内核内存池边界知识来识别动态数组。基于对象列表遍历的方法不适用于受到 DKOM 攻击的操作系统，因为 DKOM 攻击会破坏系统内核中的全局链表，使得无法遍历内存中被隐藏的内核对象。

基于签名扫描的方法通常直接从整体扫描内核内存镜像的二进制内容，在扫描过程中检查所观察到的内存子序列是否与设计的对象签名相匹配，从而确定序列所属的对象类型。相比列表遍历方法，这种方法鲁棒性更强但开销极大。Dolan-Gavitt 等[6]提出了一种针对 Windows 系统的基于值不变签名的内核数据结构检测方法，该方法根据内核数据结构的不变量生成对应数据结构的签名，但对于不存在值不变字段的数据结构难以实现签名检测。基于无监督贝叶斯学习的方法[10]从内存镜像推理数据结构布局，并使用这种布局作为数据结构的签名，该方法无需相关程序数据结构的先验知识，直接从内存镜像中学习有关数据结构的布局信息，在数据结构定义不可用的情况下仍然有效，但使用无监督学习使得该方法的检测准确度受到限制。基于规范的架构[11]根据完整性规范对内核内存数据结构进行定期检查，这些规范描述了在

内核正常执行期间必须保持的数据结构的关键语义属性，违反规范即表示存在 Rootkit。虽然该技术能检测修改控制数据结构和非控制数据结构的 Rootkits，但需要手动开发完整性规范。SigGraph[12] 根据各种内核数据结构之间的指向关系，为 Linux 内核数据结构生成了基于图的结构不变签名，相比值不变签名[6]，SigGraph 能更准确地识别具有指针字段的内核数据结构，但对每种目标数据结构都生成基于图的结构不变签名开销大，且图签名基于数据结构内部的指针映射关系，对于不含指针或指针较少的数据结构，难以生成健壮的签名，因此 SigGraph 对此类数据结构的识别精度不高。对于攻击此类内核数据结构的 DKOM 攻击，SigGraph 难以有效检测。

在分析恶意软件行为方面，chkrootkit[13] 能够面向 Linux 系统检测实施 DKOM 攻击的 Rootkit。其检测原理基于两方面：文件系统和进程。对于文件系统，chkrootkit 检查系统的文件和目录，比较它们的哈希值和已知的预期哈希值，如果文件或目录的哈希值与预期值不一致，chkrootkit 会将其标记为潜在的 Rootkit。对于进程，chkrootkit 检查系统的所有进程，通过比较它们的进程号和进程名来判断其是否是 Rootkit 创建的隐藏进程。

3　基于深度学习的 DKOM 攻击检测方案

3.1　方案概述

为了实现面向 Linux 系统的 DKOM 攻击检测，本文利用 DKOM 攻击的特性，通过判断 Linux 内存内核区域中是否存在被隐藏的内核对象来判断当前系统是否受 DKOM 攻击。为作出准确判断，需要对本文内核数据结构识别结果和现有基于对象链表遍历的内核数据结构识别结果的差异进行比较。本文基于深度学习的 Linux 系统 DKOM 攻击检测方案如图 1 所示，检测方案主要分为两个阶段。

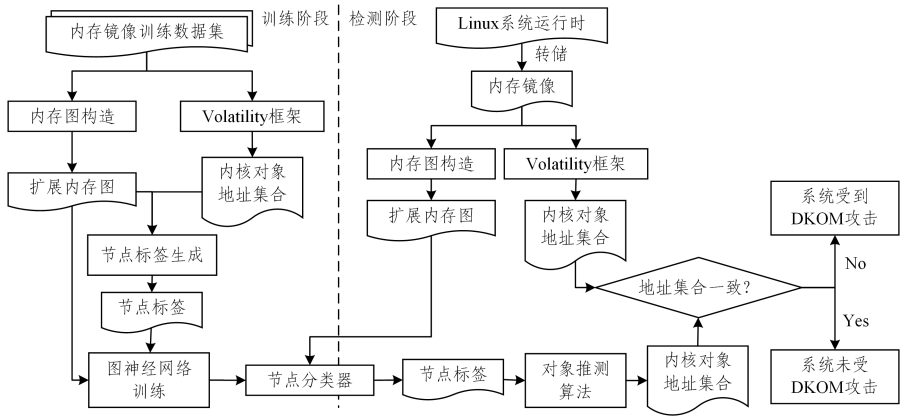

图 1　基于深度学习的 Linux 系统 DKOM 攻击检测方案流程

Fig. 1　Workflow of deep-learning based DKOM detection for Linux systems

1）训练阶段

首先，通过监督学习自动地从未受攻击的 Linux 系统的内存镜像文件中学习待检测内核对象的特征。用一系列未受攻击的 Linux 系统的内存镜像训练数据集，每一内存镜像代表 Linux 系统某一时刻的运行时内存状态。使用扩展内存图构造模块，为数据集中每一个内存镜像的内核态内存生成一种图数据结构，即扩展内存图。

其次，使用内存分析框架 Volatility[14] 对训练数据集的内存镜像进行内核对象扫描，获得内存中各种内核对象的地址与大小，记录内存镜像所包含的内核对象集合。由于训练集内存镜像无 DKOM 攻击，因此 Volatility 的分析结果可视为合理的标注依据。

然后，使用节点标签生成模块对扩展内存图中的节点分配标签。节点标签生成模块接收扩展内存图和 Volatility 分析的内核对象集合作为输入，遍历扩展内存图节点，若节点地址落入内核对象集合中某个对象的区域内，则为该节点分配对应的内核对象类型标签。节点标签信息包含了该节点所属的内核对象类型、节点在内核对象中的偏移量，以及节点自身的大小。

最后，将扩展内存图和对应的节点标签输入图神经网络模型中进行训练，最终得到一个能够满足检测精度要求的节点分类器模型。对象推测算法基于投票机制，用每个内存图节点的预测结果为内存中的特定地址上存在某种对象类型进行投票，当预测置信度超过了推测阈值时，则认为在该地址上检测到了特定类型的内核对象。因此，训练过程还会使用验证数据集搜索最优推测阈值，为对象推测算法设定合适的推测阈值。

2）检测阶段

检测系统以可能受到 DKOM 攻击的 Linux 系统内存镜像作为检测样本输入，在构造扩展内存图的基础上，使用已训练好的节点分类器模型对扩展内存图进行节点分类，为每个图节点预测出节点标签。使用最优推测阈值配置的对象推测算法，从节点分类结果推测出内存镜像中的目标内核对象地址集合。以上训练和检测功能构成了本文方案的内核对象检测系统。

使用内存分析框架 Volatility 对检测样本内存镜像进行基于对象链表遍历的传统方法分析，得到未隐藏的内核对象地址集合。本文内核对象检测系统预测的内核对象若未被 DKOM 攻击恶意隐藏，则应能被 Volatility 从内存镜像中找到。对比本文预测的内核对象地址集合与 Volatility 分析的

未隐藏内核对象地址集合的差异，若存在被隐藏的对象，则判定当前 Linux 系统受 DKOM 攻击。

以下就检测方案的关键步骤分别进行介绍。

3.2 扩展内存图定义

内存镜像包含了计算机物理内存中的所有内容，其中存在大量内核对象，大多数内核对象中都存在指针，由指针联系各个内核对象及内核对象的各个部分，因此内存镜像中有许多表示指针的字节序列，将其简称为指针字段。由于内存镜像中包含大量的指针字段，而指针来源于内核数据结构，因此指针字段的分布在很大程度上包含了内核对象在内存中的分布信息；另一方面，对于指针较少或不存在指针的内核数据结构，无法通过指针字段来反映它们在内存中的分布信息，但这些内核数据结构中会存在具有特定常量值的常量字段，可以将这种常量值看作“特殊指针”，因此我们定义常量指针为一种指针字段，通过这种常量信息来反映此类内核对象的特征。

通过提取内存镜像中以上两种指针字段的分布信息，我们为每一个内存镜像文件构造出一种图结构，称为扩展内存图。扩展内存图在原始的内存图[7]的基础上引入了常量指针，并设置其左右节点的指向来传播此类特殊指针的特征。

定义 1 扩展内存图是一个具有 4 类边的有向图 $G=(N,E_{ln},E_{rn},E_{lp},E_{rp})$，其中：

1）N 为节点集，每个节点 $n\in N$ 表示两个指针字段之间的一段连续内存字节；

2）E_{ln} 为左邻接边集，$(n_i,n_j)\in E_{ln}$ 当且仅当 n_i 是 n_j 的左邻节点（假设从左向右物理地址由小到大排布）；

3）E_{rn} 为右邻接边集，$(n_i,n_j)\in E_{rn}$ 当且仅当 n_i 是 n_j 的右邻节点；

4）E_{lp} 为左指针边集，$(n_i,n_j)\in E_{lp}$ 当且仅当 n_j 的左边邻接指针指向 n_i，特殊地，当 n_i 左边邻接常量指针时，定义 $(n_i,n_i)\in E_{lp}$；

5）E_{rp} 为右指针边集，$(n_i,n_j)\in E_{rp}$ 当且仅当 n_j 的右边邻接指针指向 n_i，特殊地，当 n_i 右边邻接常量指针时，定义 $(n_i,n_i)\in E_{rp}$。

图 2 为一个构建扩展内存图的案例。图 2(a)是一段原始内存，A,B,C,D 为指针字段和常量指针字段之间的内容，箭头指向的位置就是指针字段存储的地址。指针内部一般存放的是某个数据结构的起始地址，而数据结构的起始处未必是一个指针。由于我们只用指针切分出扩展内存图节点，而未使用指针指向的目标字节切分图节点，因此指针指向的位置可能在某个节点内部。对于图 2(a)中的原始内存，根据扩展内存图的定义，节点集 N 包括 A,B,C,D；E_{ln} 包括 $A\rightarrow B$，$B\rightarrow C$，$C\rightarrow D$；E_{rn} 包括 $B\rightarrow A$，$C\rightarrow B$，$D\rightarrow C$；E_{lp} 包括 $C\rightarrow B$，$D\rightarrow C$，$D\rightarrow D$；E_{rp} 包括 $C\rightarrow A$，$D\rightarrow B$，$C\rightarrow C$。综上所述，图 2(a)所对应的扩展内存图如图 2(b)所示。图 2(c)为一种多个指针字段连续出现的特殊情况，在 A 和 B 之间有两个连续的指针，分别指向 C 和 D。此时，对于节点 A，将其右边的两个连续指针都视为其右边界指针，创建由节点 C 和节点 D 指向节点 A 的 rp 类型的有向边 $C\rightarrow A$ 和 $D\rightarrow A$。同理，对于节点 B，创建 lp 类型的有向边：$C\rightarrow B$ 和 $D\rightarrow B$。

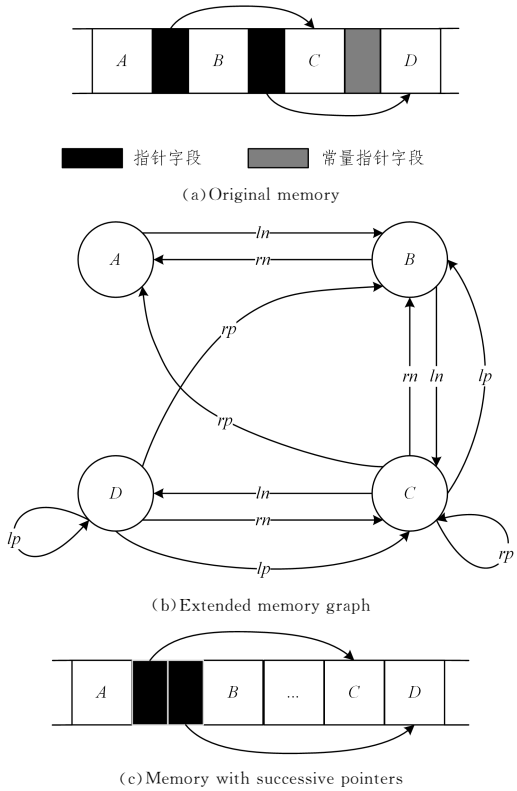

图 2 扩展内存图定义

Fig. 2 Definition of extended memory graph

基于上述扩展内存图定义，我们进行 Linux 系统内存镜像的扩展内存图构造。首先，扩展内存图构造模块接收一个内存镜像二进制文件，通过 Volatility 分析得到该内存内核地址空间对应的物理地址范围。然后，遍历内存镜像中这些物理地址范围内的内容，判断按指针长度逐一读取的字段值是否属于内核地址空间范围或属于常量指针集合 *cptrs*，若是，判断该字段为指针，记录所有指针字段的存储地址与它指向的地址，得到指针自身地址集合 *ptrs* 与指针指向地址的映射关系 *ptr2dest*。之后，再根据指针自身地址集合 *ptrs* 的内容，对内存镜像的内核空间字节序列进行划分，将划分得到的字节序列作为扩展内存图的节点，记录它们的起始地址与内容，得到节点集 N。定义节点大小 $size(n)$ 为节点 n 的字节数。最后，扩展内存图构造模块使用算法 1，根据节点集 N、指针地址集合 *ptrs* 以及指针地址映射 *ptr2dest*，生成边集 E_{ln}，E_{rn}，E_{lp}，E_{rp}。

算法 1 构建扩展内存图边集 E_{ln}，E_{rn}，E_{lp}，E_{rp}

输入：节点集 N，指针集合 ptrs，指针地址映射 ptr2dest，常量指针集合 cptrs

输出：边集 E_{ln}，E_{rn}，E_{lp}，E_{rp}

```
1. E_ln←∅, E_rn←∅, E_lp←∅, E_ln←∅;
2. for i←1 to len(ptrs)-1 do
3.     E_ln[N[i]].add(N[i-1]);//更新左邻接边集
4.     E_rn[N[i-1]].add(N[i]);//更新右邻接边集
5.     updateEdgeSet(-1,N[i]);//更新节点 i 的左指针边集
6.     updateEdgeSet(1,N[i]);//更新节点 i 的右指针边集
7. end for
8. Procedure updateEdgeSet(dir,n);// n 为节点
```

```
9.    E←(dir=−1)? E_lp[n]: E_rp[n];//选择需要更新的边集，dir 取
      −1 代表左，1 代表右
10.   ptr←n +dir · word_sz;//当前节点左或右侧的地址
11.   while ptr∈ptrs do //若 ptr 是指针地址
12.     if ptr∈cptrs then //ptr 是常量指针
13.       E[n]. add(n);
14.     else
15.       ptr_dest←ptr2dest(ptr);
          //获取当前指针指向的地址
16.       if ptr_dest在节点 x 的范围且 x∈N then
          //若指向某个节点 x
17.         E[n]. add(x);
18.       end if
19.     end if
20.     ptr←ptr+dir · word_sz; //寻找连续指针
21.   end while
22. end Procedure
```

算法 1 的主体(第 1－7 行)迭代调用过程 updateEdgeSet(第 8－22 行)完成边集构造。算法 1 复杂度为 $|ptrs|^2$，因而实际运行开销取决于内核地址空间中找到的指针字段个数。但由于在内存镜像的瞬时运行态，单个指针字段包含固定值，因此每一类边的数量均受指针字段数量上限的约束，使得实际扩展内存图的复杂度可接受。

3.3　图节点标签构造

由于本文使用监督学习，因此需事先对训练数据进行标记。为构造扩展内存图节点标签，节点标签生成模块接收一个扩展内存图 $G=(N,E_{ln},E_{rn},E_{lp},E_{rp})$ 以及该扩展内存图对应内存镜像的内核对象集合 O 作为输入，输出一个节点标签集合。其中内核对象集合 O 由 Volatility 框架对内存镜像分析得到，包含了内存镜像内部所有内核对象的类型、起始地址以及大小。节点标签生成模块遍历节点集 N。在遍历的过程中，判断当前节点地址是否属于内核对象集合 O 中某个对象的一部分，如果属于，则为当前节点生成与该对象类型相关的三元组标签，否则为当前节点分配 *NULL* 标签。当节点属于某个内核对象的一部分时，为其生成的节点标签形如“对象类型_节点偏移_节点大小”。其中，对象类型指该节点所属的内核对象的类型，节点偏移指节点所属内核对象的起始地址与节点起始地址的差值，节点大小指节点的实际字节数。图 3 展示了一个 task_struct 对象的节点标签，如 task_56_16 表示该节点是某个 task_struct 对象的一部分，且距离 task_struct 对象起始地址偏移量为 56 字节，该节点的大小为 16 字节，其他节点同理。同时，图 3 中的 3 个节点标签所示的节点偏移均指向同一个 tast_struct 对象的起始地址。

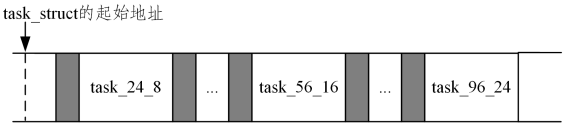

图 3　一个 task_struct 对象的节点标签示例

Fig. 3　Node label example of a tast_struct object

一个内核对象内部可能包含很多个图节点，即该对象可能包含一些稀有的节点标签，这类标签在内核对象中出现的概率很低，稀有的节点对推断对象类型不利，因此，为得到健壮的检测模型，需要筛除一些离群的节点标签，只保留一个对象中的高频节点标签。将高频节点称为关键节点，c 类型内核对象的关键节点标签集合记为 $L(c)$，具体实现中，取出现次数最频繁的前 20 个节点标签构成 c 类型对象的 $L(c)$ 集合，用于对象推测和推测阈值的选择。

3.4　图神经网络模型

本文方案的图神经网络模型 M 由嵌入网络 F_{w1} 和分类器网络 G_{w2} 组成，因而有 $M=G_{w2}(F_{w1}(\cdot))$。图神经网络的嵌入网络部分由关系图卷积网络(Relational Graph Convolutional Networks，RGCN)[15-16] 模型实现，该模型以扩展内存图作为输入，通过多次迭代后为每个节点聚合它相邻节点的信息，得到图中所有节点的低维的嵌入向量；而分类器网络使用全连通网络模型，用于将嵌入网络的输出节点嵌入向量进行分类，得到每个节点预测标签。相比传统的图卷积网络(GCN)对边类型建模的限制，RGCN 能够处理具有多种边类型的异构图，因而适用于本文扩展内存图的嵌入，并能解决节点分类问题。

3.4.1　嵌入网络

嵌入网络的功能是将扩展内存图中的每个节点映射为一个低维的嵌入向量，在映射过程中最大化保留节点周围的拓扑信息。本文设计的扩展内存图是一种节点同构、边异构的图，具有 4 种类型的边。定义扩展内存图中的任一节点 i 的特征由一个 d 维向量 $\boldsymbol{x}_i$ 表示，其中每个维度为节点 i 中一个内存字节。由于扩展内存图节点在内存镜像中的长度不一，因此若节点 i 比 d 字节长，则对其从地址低位截断，只保留地址低位前 d 字节；若节点 i 比 d 字节短，则用 0 值填充其地址高位到 d 字节长度。

嵌入网络 F_{w1} 的输入是节点的向量表示 $\boldsymbol{x}_i$，输出为嵌入向量 $\boldsymbol{h}_i$。对于扩展内存图中的任一节点，嵌入网络需要从该节点的所有邻居节点获得它周围的拓扑信息，故定义节点 i 的嵌入向量 $\boldsymbol{h}_i$ 的计算公式如式(1)所示：

$$\boldsymbol{h}_i=F_{w1}(\boldsymbol{x}_i,\boldsymbol{x}_{E_{ln}[i]},\boldsymbol{x}_{E_{rn}[i]},\boldsymbol{x}_{E_{lp}[i]},\boldsymbol{x}_{E_{rp}[i]}) \tag{1}$$

RGCN 定义了如式(2)所示的传播模型：

$$\boldsymbol{h}_i^{(l+1)}=\sigma(\boldsymbol{W}_0^{(l)}\boldsymbol{h}_i^{(l)}+\sum_{r\in R}\sum_{j\in N_i^r}\frac{1}{c_{i,r}}\boldsymbol{W}_r^{(l)}\boldsymbol{h}_j^{(l)}) \tag{2}$$

其中，$\sigma(\cdot)$ 表示激活函数，$\boldsymbol{h}_i^{(l)}$ 表示节点 i 的第 l 层嵌入表示，$\boldsymbol{h}_{(j)}^{l}$ 表示节点 i 的所有邻居节点在第 l 层嵌入表示，N_i^r 表示与节点 i 的关系为 r 的邻居节点的集合。$c_{i,r}$ 是一个归一化常数，和与之相乘的函数组成概率密度函数，主要作用是提高模型的收敛速度，可以手动设置也可以学习得到。$\boldsymbol{W}_r^{(l)}$ 是与节点 i 的关系为 r 的邻居节点在第 l 层的权重矩阵，特别地，$\boldsymbol{W}_0^{(l)}$ 是节点 i 自身在第 l 层的权重矩阵，$\boldsymbol{W}_r^{(l)}$ 通过监督学习来获得。RGCN 网络通过对节点不同的邻居分配不同的权重矩阵，进而实现对边异构图的特征学习。

结合 RGCN 的传播模型，引入不同层数节点嵌入向量之间的关系，第 $l+1$ 层的嵌入向量依赖于第 l 层的嵌入向量，因而式(1)可以转化为式(3)：

$$\boldsymbol{h}_i^{(l+1)}=F_{w1}(\boldsymbol{h}_i^{(l)},\boldsymbol{h}_{E_{ln}[i]}^{(l)},\boldsymbol{h}_{E_{rn}[i]}^{(l)},\boldsymbol{h}_{E_{lp}[i]}^{(l)},\boldsymbol{h}_{E_{rp}[i]}^{(l)}) \tag{3}$$

对于任一节点，嵌入网络会以 BFS(广度优先搜索)的方式收集关于其邻居节点的信息。我们将每个节点 i 的初始信息 $\boldsymbol{x}_i$ 作为它第 0 层的嵌入向量 $\boldsymbol{h}_i^{(0)}$，然后进行 K 次迭代，在第 l 次迭代中，每个节点遍历自己的邻居节点，并将邻居节点的第 $l-1$ 层嵌入向量聚合到自身的嵌入向量 $\boldsymbol{h}_i^{(l)}$ 中。执行完 K 次迭代后，嵌入网络节点 i 的最终嵌入向量 $\boldsymbol{h}_i^{(K)}$。最终将式(3)实现为式(4)：

$$\begin{aligned}\boldsymbol{h}_i^{(k+1)}=\mathrm{ReLU}(&\boldsymbol{W}_0^{(k)}\boldsymbol{h}_i^{(k)}+\sum_{j\in E_{ln}[i]}\frac{1}{c_1}\boldsymbol{W}_1^{(k)}\boldsymbol{h}_j^{(k)}+\\&\sum_{j\in E_{rn}[i]}\frac{1}{c_2}\boldsymbol{W}_2^{(k)}\boldsymbol{h}_j^{(k)}+\sum_{j\in E_{lp}[i]}\frac{1}{c_3}\boldsymbol{W}_3^{(k)}\boldsymbol{h}_j^{(k)}+\\&\sum_{j\in E_{rp}[i]}\frac{1}{c_4}\boldsymbol{W}_4^{(k)}\boldsymbol{h}_j^{(k)})\end{aligned} \tag{4}$$

其中，权重矩阵 $\boldsymbol{W}_i(i=0,1,2,3,4)$ 是节点用于聚合自身以及邻居节点特征的权重参数，由于 $\boldsymbol{W}_i(i=0,1,2,3,4)$ 是 5 个完全独立的权重矩阵，因此 RGCN 网络对不同类型邻居的特征传播方式是不同的。

3.4.2 分类器网络

分类器网络接收嵌入网络的输出，即节点的嵌入向量，输出为节点的预测标签。假定所有节点标签的集合为 L，分类器网络完成节点嵌入向量到节点标签向量的映射为 G_{w2}，使得 $l_i=G_{w2}(\boldsymbol{h}_i)$，其中 $i\in N, l_i\in L$。本文选择用全连接网络模型实现分类器网络，该模型具有多层隐藏层，每个隐藏层的神经元都通过 ReLU 激活函数实现非线性的分类，随后是实现输出最终分类的 softmax 层。

3.4.3 图神经网络训练

本文提出的图神经网络的训练过程如下：针对多种内核对象类型分别实现分类器，在有标记的训练数据集内存镜像上，对于不同类型的分类器，使用对应类型的标记数据进行训练。在单个分类器训练过程中，将训练样本输入嵌入网络，收集上下文信息。在传播 K 次迭代后，最终嵌入向量被输入分类器网络以生成预测输出标签。为了训练图神经网络模型的权重，我们计算预测标签和标注标签之间的交叉熵损失，并在最小化误差的过程中更新权重，更新的参数包括嵌入网络的 w_1(包括权重矩阵 $\boldsymbol{W}_i(i=0,1,2,3,4)$ 的网络参数和归一化常数 c_i 的值)和分类器网络的 w_2。定义训练数据集为 $D=\{d_1,d_2,\cdots\}$，其中每个 $d_i=(x^{(i)},y^{(i)})$ 是一对节点向量与对应的节点标签向量，优化目标函数 $J(w_1,w_2)$ 如式(5)所示：

$$J(w_1,w_2)=\arg\min_{w_1,w_2}(\sum_{i=1}^{|\mathcal{D}|}\mathcal{L}(\boldsymbol{y}^{(i)},M(\boldsymbol{x}^{(i)}))) \tag{5}$$

其中，$\mathcal{L}(\cdot)$ 为交叉熵损失函数，$\boldsymbol{x}^{(i)}$ 为节点的初始向量，$M(\boldsymbol{x}^{(i)})$ 即图神经网络模型对 $\boldsymbol{x}^{(i)}$ 的预测标签 $\hat{\boldsymbol{y}}^{(i)}$。使用反向传播算法[17]实现最小化误差，得到充分训练的节点分类器及最优参数，训练过程使用 PyTorch 实现。

3.5 对象推测算法

对象推测算法将图神经网络模型输出的各个节点的预测标签转化为对内核对象起始地址的预测，如果多个扩展内存图节点的预测标签在转化后均指明在内核地址 a 处存在一个 c 类型的内核对象，即可确信从地址 a 起始存在一个 c 类型对象。因此，节点的预测标签可视为对于对象存在和对象类型的一次投票。例如，预测标签为 task_56_16 的节点，其标签可视为投票选择从该节点起始地址向前偏移 56 字节处存在一个 task_struct 对象。若图 3 中的 3 个节点的标签是由图神经网络模型预测出的，则可认为它们在为同一 task_struct 投票。

由所有节点预测标签可以生成一组候选对象地址 $A=\{a_1,a_2,\cdots\}$ 以及每个地址对应的投票节点集合。接下来需要确定一个地址 $a\in A$ 是否确实是某一对象的起始地址。理想情况下，一个内核对象的范围内，c 类型的所有关键节点都投票支持唯一的 a 成为 c 类型对象的地址，对象推测算法即确定在地址 a 上检测到一个 c 类型对象。然而，由于训练误差等因素，可能仅小部分关键节点标签投票支持地址 a，因而报告地址 a 为 c 类型的信心会降低。

类似于文献[7]，本文使用一种投票频率与投票标签数加权的投票机制，给不同的节点标签赋予不同的投票权重。本文定义预测函数 $\lambda(a,c)$ 如式(6)所示：

$$\lambda(a,c)=\begin{cases}1, & \sum_{l_i\in L(a,c)}(p(c,l_i)+|L(a,c)|-Q)>\delta\\0, & \text{otherwise}\end{cases} \tag{6}$$

其中，关键节点标签 l_i 的权值 $p(c,l_i)$ 表示 c 类型对象中被 l_i 标签投票的比例；$L(a,c)$ 为所有为地址 a 投票为 c 类型的关键节点标签的集合。预测置信度定义为，若将地址 a 投票为 c 类型的关键节点标签的数量超过 Q 更多，且 c 类型对象中被这些关键节点标签投票的比例更高，则预测地址 a 为 c 类型对象的置信度越高。我们通过计算预测置信度和预定义阈值 δ 之间的差异来判断是否在地址 a 检测到 c 类型对象，当加权投票的置信度值超过阈值 δ 即 $\lambda(a,c)=1$ 时，表明在地址 a 检测到 c 类型对象；$\lambda(a,c)=0$ 表明未在地址 a 检测到 c 类型对象。易见，当 $\delta>0$，将地址 a 投票为 c 类型的标签数量小于 Q 将直接导致无法将地址 a 检测为 c 类型对象，从而达到鼓励更多的节点标签投票达成共识的效果。

当节点分类器模型训练完成后，使用验证数据集搜索最优推测阈值 δ，为对象推测算法设定合适的推测阈值。一方面，依据验证数据集的 Volatility 分析结果作为内核对象地址集合的预期结果；另一方面，为验证数据集构造扩展内存图，输入节点分类器进行节点预测，得到预测的节点标签集合，以初始推测阈值 $\delta=0$ 起始，迭代应用对象推测算法(式(6))得到当前推测阈值 δ 下的内核对象推测结果，比较推测结果和预期结果的差异，计算 F1-score 值，多次迭代调整推测阈值 δ，获得最高 F1-score 值对应的阈值 δ 作为最优推测阈值。检测阶段使用节点分类器配合最优推测阈值进行内核对象地址推测，得到一组目标对象地址，即内核对象检测的最终结果。

3.6 DKOM 攻击检测

本文提出的对象检测系统能够从 Linux 内存镜像二进制文件中识别出预期的所有内核对象。正常的内核对象会存在于 Linux 系统内核对象链表中，将这些正常对象记为集合 pos；被 DKOM 攻击恶意隐藏的内核对象则不存在于内核对象链表中，将这些被攻击的对象记为集合 neg。本文内核对

象检测系统的识别结果的最优目标是 $pos \cup neg$，但本文检测系统不能区分其中的两类内核对象，因此借助了 Volatility 框架。Volatility 在检测内核对象时，遍历系统内存中内核对象链表结构，因此可以检测到 pos 集合的内容。用本文的对象检测系统和 Volatility 框架对同一 Linux 系统内存镜像进行分析，对比分析结果可知 neg 是否为空集，从而判定当前系统内存中是否存在被恶意隐藏的内核对象，即系统是否受到 DKOM 攻击。

4　实验分析

4.1　实验环境与数据集

本文实验环境硬件参数为：Intel Xeon Platinum 8163 CPU @2.5 GHz，92G RAM；软件环境为：Ubuntu16.04 64 位，Volatility 2.6，Python3.7.3，PyTorch1.7.1。

本文实验数据集由 Ubuntu 系统内存镜像文件组成，Ubuntu 以客户机形式运行在 VMware 上，利用 VMware 提供的获取虚拟机快照功能获得 Linux 镜像文件。为实现自动收集大量不同的内存镜像，本文开发了一个脚本工具，实现自动重启 Ubuntu 客户机、在开机后模拟用户随机操作并自动收集系统某运行时刻内存镜像的功能。本文实验最终使用以下两类内存镜像数据集：

1）内存镜像数据集 D1：包含 500 个 Linux 内存镜像，其中每个内存镜像均从未被 DKOM 攻击的系统中提取。提取步骤为：(1)脚本工具启动在 VMware 中的 Ubuntu 客户机，待客户机启动后，脚本控制 Linux 系统自动触发 20～30 个随机操作，包括启动常用软件、打开流行网站网页、随机打开办公文档、图片文件等；(2)等待 1 min，脚本控制 VMware 对虚拟机内存进行快照并转储到主机系统外存；(3)脚本控制客户机重启并重复上述步骤，直至收集到足够的内存镜像作为实验的数据集。D1 中每个内存镜像大小为 1GB。将这些内存镜像按照 6∶2∶2 的比例随机划分为训练数据集、验证数据集和测试数据集。表 1 列出了由 Volatility 框架分析得出的一个未受到攻击的 Linux 系统内存镜像中的内核对象统计信息，相应扩展内存图的 4 类边数量均在 178 万～182 万条之间，节点数约为 180.9 万个。进一步统计本文扩展内存图引入常量指针所导致的节点数和变数的增长可以发现，相比文献[7]定义的内存图，常量指针导致节点数增加 11 358 个，约占0.63%，边数量增加 26 732 条，约占 0.37%。虽然常量指针引入的新节点和边的占比不大，但对于特定类型的缺乏指针的小内核数据结构，其引入的新特征已足够明显，能够有效帮助识别特定内核数据结构(见第 4.2.2 节)。

表 1　内存转储中的内核对象统计信息

Table 1　Kernel object statistics of a memory dump

内核对象类型	对象长度	数量
task_struct	5440	529
module	896	68
file	256	1314
dentry	192	1053
inode	560	781
module_attribute	56	68
module_sect_attr	72	68

2）内存镜像数据集 D2：包含 250 个 Linux 内存镜像，其中每个内存镜像是从受到 DKOM 攻击的 Linux 系统中提取得到的，用于本文方案对 DKOM 攻击检测进行测试。我们使用了 5 个现实世界的基于 DKOM 攻击的内核级 Rootkit 来进行攻击，它们分别是 adore-ng[18]，Wukong[19]，KBeast[20]，Jynx2[21]和 LilyOfTheValley[22]。为模拟 Linux 系统受 DKOM 攻击时的状态，本文在前述自动执行脚本中加入了启动以上 Rootkit 的步骤，所使用 Rootkit 均进行了 DKOM 攻击，因此每个内存镜像样本均包含由 DKOM 攻击隐藏的系统内核对象。在单个 Linux 内存镜像上仅应用一种 Rootkit，每一种 Rootkit 通过 DKOM 攻击隐藏若干个目标内核对象。由于知道每次 DKOM 攻击针对的具体内核对象信息，故容易计算后续实验所需的 ground truth。

4.2　结果与分析

4.2.1　内核对象检测有效性

本文方案通过内核对象识别的方式来实现 DKOM 攻击检测，因此最终 DKOM 攻击检测的有效性取决于内核对象检测系统的有效性。本小节使用数据集 $D1$，将训练数据集、验证数据集和测试数据集的内存镜像文件输入扩展内存图构造模块，构造出 500 个扩展内存图。将所有扩展内存图输入节点标签构造模块，即可得到每个扩展内存图对应的节点标签集合。将训练数据集和验证数据集中的所有扩展内存图和对应的节点标签集合输入图神经网络训练过程和调参过程。以检测 tast_struct 的图神经网络模型为例，以上过程得到的图神经网络模型的超参数最优取值如表 2 所列。这些最优超参数取值需在不同超参数取值组合下对有效性指标 F1-score 进行网格搜索得到，具体搜索过程如表 3 所列。

将测试数据集对应的扩展内存图输入训练好的对象检测系统，对 task_struct 和 module 等 7 种常见内核数据结构进行检测。表 4 列出了本文提出的内核对象检测系统的检测有效性。可见，本文对象检测系统对 task_struct，module，module_sect_attr 结构对象的检测精度非常理想，Precision 和 Recall 均达到 99%以上；对 file 和 module_attribute 结构对象检测精度较好；而对 dentry 和 inode 结构对象的检测精度较差是因为训练过程使用的 Volatility 标注未能识别出无 DKOM 攻击的 Linux 内存镜像中的所有 dentry 和 inode 结构，导致对扩展内存图中这两类结构的关键节点标记不全，最终影响了图神经网络的训练效果。

表 2　task_struct 检测用图神经网络的超参数及最优值

Table 2　Optimal hyper-parameters of RGCN for detecting task_struct

超参数	含义	最优值
L_F	嵌入网络 F 的神经网络层数	3
L_G	分类器网络 G 的神经网络层数	3
K	RGCN 每个节点在聚合邻居迭代的跳数	4
vector_size	节点初始向量的维度	64
embedding_size	嵌入网络输出的节点嵌入向量维度	32
vector_type	节点初始向量填充方式	padding
keep_prob	随机失活比率，用于防止过拟合	0.8

表 3 task_struct 检测用图神经网络的超参数最优值搜索

Table 3 Searching of optimal hyper-parameters of RGCN for detecting task_struct

超参数	取值	*F1-score*
L_F	1	0.6418
	2	0.9661
	3	**0.9914**
	4	0.9185
L_G	1	0.4964
	2	0.8676
	3	**0.9914**
	4	0.9593
K	1	0.7035
	2	0.9263
	3	0.9584
	4	**0.9914**
	5	0.9732
vector_size	16	0.7354
	32	0.8699
	64	**0.9914**
	128	0.9362
embedding_size	16	0.9560
	32	**0.9914**
	64	0.9744
vector_type	repeat	0.9363
	padding	**0.9914**
keep_prob	0.6	0.8893
	0.7	0.9722
	0.8	**0.9914**
	0.9	0.9491

表 4 内核对象检测有效性

Table 4 Effectiveness of kernel object detection

内核对象类型	*Precision*	*Recall*	*F1-score*
task_struct	0.9904	0.9923	0.9914
module	1.0000	0.9970	0.9985
file	0.9784	0.9683	0.9733
dentry	0.9256	0.9650	0.9448
inode	0.9342	0.9168	0.9256
module_attribute	0.9793	0.9834	0.9813
module_sect_attr	1.0000	0.9961	0.9978

4.2.2 扩展内存图性能分析

本文提出的扩展内存图在 DeepMem[7] 的内存图的基础上加入了常量字段及相应的图指针表示，提升了对缺少指针字段但存在常量字段的内核对象结构信息的表达能力。因此，本文提出的图神经网络模型能够从扩展内存图中学习到这一类内核对象的特征信息，提高对象检测模型对此类内核数据结构的检测精度。

为对比本文扩展内存图与 DeepMem 内存图的特点，使用 Linux 内核中的一种缺少指针但存在常量字段的数据结构 module_attribute 来进行实验，module_attribute 的结构体定义如下：

```
struct module_attribute {
  struct attribute attr;
  ssize_t( * show)(struct module_attribute * ,
        struct module * ,char * );
  ssize_t( * store)(struct module_attribute * ,
        struct module * ,const char * ,size_t);
};
```

从结构体定义中可见，module_attribute 数据结构较小，内部指针字段很少，但其内部包含的结构体 attribute 有常量字段，本文提出的扩展内存图将其作为常量指针，而 DeepMem 的内存图则忽略此常量特征。由于 DeepMem 实现[7,23]仅适用于 Windows 内核对象，无法直接使用，因此，本文针对 Linux 的 module_attribute 内核对象类型，复现了 DeepMem 内存图实现，并按流程训练了 DeepMem 的内核对象识别功能。表 5 列出了 module_attribute 结构分别在 DeepMem 内存图和本文扩展内存图中所包含的关键节点标签。由表 5 可见，与本文扩展内存图得出的关键节点标签集相比，DeepMem 内存图得出的关键节点标签 module_attribute_0_24 在本文扩展内存图中变为了关键节点标签 module_attribute_8_16，且 module_attribute_8_16 的权值比 module_attribute_0_24 的权值更大，有利于图神经网络模型学习内核对象结构信息。

表 5 module_attribute 在不同图结构中的关键节点标签

Table 5 Critical node label of module_attribute in different graph structures

DeepMem 内存图		本文扩展内存图	
节点标签	出现频率	节点标签	出现频率
32_24	0.2794	32_24	0.2794
0_24	0.0735	8_16	1.0000
32_40	0.7206	32_40	0.7206

表 6 列出了使用 DeepMem 内存图与本文提出的扩展内存图分别得出的 module_attribute 结构对象的图神经网络模型检测有效性对比结果。可以看出，使用 DeepMem 内存图结构训练的模型，对于 module_attribute 检测的 Precision 和 Recall 均明显低于使用本文扩展内存图训练的模型。因此本文提出的扩展内存图相比 DeepMem 内存图能更好地提取指针较少的内核对象的内部特征。

表 6 使用不同图结构的对象检测系统对 module_attribute 的检测有效性

Table 6 Detection effectiveness of different memory graphs on module_attribute

内存图结构	*Precision*	*Recall*	*F1-score*
DeepMem 内存图	0.4640	0.6321	0.5352
扩展内存图	0.9793	0.9834	0.9813

4.2.3 DKOM 攻击检测性能测试

在 Linux 系统 DKOM 攻击检测方面，与本文方法相关的工作包括文献[11-13]，但文献[11]和[12]未公开实现，因此，本小节将本文 DKOM 攻击检测方案与 DKOM 攻击检测工具 chkrootkit[13] 进行比较。chkrootkit 是常用的 Linux 系统 DKOM 攻击实时检测工具，将其安装在待测系统后，它通过扫描目标系统，检测内核中是否存在被 DKOM 攻击隐藏的文件或进程。

本小节使用数据集 $D2$ 和从数据集 $D1$ 中随机选择的 250 个未受 DKOM 攻击的内存镜像样本进行实验。对这些内存镜像，首先利用本文方案进行分析，检测 DKOM 攻击。然后，作为公平对比，在 Ubuntu 客户机中安装 chkrootkit 工具，分别在 250 个安装了前述 5 个现实世界内核级 Rootkit 的 Ubuntu 客户机上进行隐藏进程、隐藏模块等 DKOM 攻击。在 DKOM 攻击后，使用 chkrootkit 对客户机系统进行在线恶

意 DKOM 扫描，记录检测结果。

在检测的有效性方面，分别使用本文方案和 chkrootkit 对每个内存镜像样本进行检测，将实际受到 DKOM 攻击的样本记为 ground true positive，记录所有检测结果并计算检测的 Precision，Recall 及 F1-score 的值，最终结果如表 7 所列。具体分析检测结果发现，本文方案除了未检测出 Wukong 这种 Rootkit 隐藏 file 结构的 DKOM 攻击，其他的被 DKOM 攻击隐藏的内核数据结构均被无漏报地检测出来，且误报个数明显少于 chkrootkit。chkrootkit 对 KBeast 的检测效果优于对其他 4 种 Rootkit 的检测效果。实验结果表明，本文 DKOM 检测方案的有效性高于 chkrootkit 工具。检测的精确度提高 20.1%，召回率提高 32.4%。

表 7　相比 chkrootkit 的 DKOM 攻击检测有效性

Table 7　DKOM attack detection effectiveness compared with chkrootkit

Metric	chkrootkit	本文方案
TP	158	239
FP	63	22
FN	92	11
Precision	0.7149	0.9157
Recall	0.6320	0.9560
F1-score	0.6709	0.9354

在 DKOM 攻击检测的时间开销方面，对比本文方案的检测过程运行时开销与 chkrootkit 工具的 Rootkit 在线扫描时间。不计入本文图神经网络模型训练开销的原因是这一过程为离线的过程，且一次训练结果可应用于任意多个 Linux 内存镜像。结果显示，本文检测方案对单个内存镜像的检测时间平均开销(2 min 28 s)高于 chkrootkit 工具对单个 Linux 系统的平均在线扫描时间(35 s)。这是因为本文方案是一种静态分析方法，从系统内存的二进制内容进行分析，检测更加彻底，也更耗时。而 chkrootkit 实时检测仅扫描文件系统及运行的进程来进行检查，这种方法虽然检测速度快，但存在安全隐患：Rootkit 可能针对 chkrootkit 进行攻击，以防止 chkrootkit 检测到自身存在，如修改系统配置文件以禁止 chkrootkit 运行或阻止 chkrootkit 显示检测结果。而本文方案以分析待测系统内存镜像的方式检测 DKOM 攻击，检测方案与系统运行环境完全隔离，因此本文方案不会有这种安全隐患。

5　讨论

本文方案是 Linux 内核版本敏感的。本文实验结果在与特定 Rootkit 适配的 Linux 内核版本下测得，但不失一般性地，本文方案还可以支持 Volatility 框架所支持的其他 Linux 内核版本。将本文工具迁移至其他 Linux 内核版本时，可能会由于特定内核数据结构在不同内核版本间的演化，检测结果存在差异。因此，应根据不同内核版本训练特定的节点分类器模型。

同时，本文方案已在表 1 所列的 7 种内核数据结构上证明有效，能够检测出的 DKOM 攻击也仅限于对这些类型的内核数据结构进行操纵的攻击。由于内核数据类型众多，本文方案能否适用于其他类型的内核数据结构及操纵其他类型内核数据结构的 DKOM 攻击，尚待进一步确认。

结束语　本文实现了一种基于深度学习的 Linux 系统 DKOM 攻击检测方案。该方案结合了内存取证与和图神经网络技术，实现了一种基于扩展内存图和图神经网络模型的内核对象检测系统，进一步通过对比检测出的内核对象与 Volatility 框架识别出的 Linux 内核对象链表中所有对象的差异，判断 Linux 系统是否受到 DKOM 攻击，解决了现有 DeepMem 方案无法识别 Linux 系统内核对象且对缺少指针字段的小内核对象识别能力有限的问题。相比现有 Linux 系统，DKOM 检测工具 chkrootkit 实现了更高的检测精度和安全性。

实验发现，对特定 Rootkit 注入的针对 file 结构对象的隐藏，本文方案的检测效果不明显，结合在数据集 *D*1 上得出的内核对象检测有效性结果可知，对于 file，inode 和 dentry 结构的内核对象，DKOM 的检测精度还有待进一步提高。未来的主要工作是针对这些类型内核结构，改进基于 Volatility 的图节点标注的准确性；另一个潜在应用是将本文图神经网络模型用于在转储的 Linux 内核镜像的网络栈缓冲区中识别潜在的网络攻击特征，从而实现与现有在线检测[24]不同的离线检测。

参 考 文 献

[1] JOY J，JOHN A，JOY J. Rootkit detection mechanism：A survey [C]// Proceedings of International Conference on Parallel Distributed Computing Technologies and Applications. Berlin：Springer，2011：366-374.

[2] BUTLER J. Direct Kernel Object Manipulation [EB/OL]. https://www. blackhat. com/presentations/win-usa-04/bh-win-04-butler. pdf.

[3] YIN H，SONG X，EGELE M，et al. Panorama：Capturing System-Wide Information Flow for Malware Detection and Analysis [C]// Proceedings of the 14th ACM Conference on Computer and Communications Security. New York：ACM，2007：116-127.

[4] KRUGEL C，ROBERTSON W，VIGNA G. Detecting Kernel-Level Rootkits through Binary Analysis [C]// Proceedings of the20th Annual Computer Security Applications Conference. Piscataway：IEEE，2004：91-100.

[5] BALIGA A，GANAPATHY V，IFTODE L. Automatic Inference and Enforcement of Kernel Data Structure Invariants [C]// Proceedings of the 24th Annual Computer Security Applications Conference. Piscataway：IEEE，2008：77-86.

[6] DOLAN-GAVITT B，SRIVASTAVA A，TRAYNOR P，et al. Robust Signatures for Kernel Data Structures [C]// Proceedings of the 2009 Conference on Computer and Communications Security. New York：ACM，2009：566-577.

[7] SONG W，YIN H，LIU C，et al. DeepMem：Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis [C]// Proceedings of the 2018 Conference on Computer and Communications Security. New York：ACM，2018：606-618.

[8] CARBONE M，CUI W，LU L，et al. Mapping Kernel Objects to Enable Systematic Integrity Checking [C]// Proceedings of the 16th ACM Conference on Computer and Communications Secu-

rity. New York: ACM, 2009: 555-565.
[9] LIN Z, ZHANG X, XU D. Automatic Reverse Engineering of Data Structures from Binary Execution [C]// Proceedings of the Network and Distributed System Security Symposium. The Internet Society, 2010: 1-18.
[10] COZZIE A, STRATTON F, XUE H, et al. Digging for Data Structures [C]// Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation. USENIX Association, 2008: 255-266.
[11] PETRONI J N, FRASER T, WALTERS A, et al. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data [C]// Proceedings of the 15th USENIX Security Symposium. USENIX Association, 2006: 289-304.
[12] LIN Z, RHEE J, ZHANG X, et al. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures [C]// Proceedings of the Network and Distributed System Security Symposium. The Internet Society, 2011: 1-18.
[13] MURILO N, STEDING-JESSON K. chkrootkit: Locally Checks for Signs of a Rootkit[EB/OL]. http://www.chkrootkit.org/.
[14] The Volatility Foundation. Volatility Framework- Volatile Memory Extraction Utility Framework [EB/OL]. (2020-12-11) [2023-04-03]. https://github.com/volatilityfoundation/volatility.
[15] THANAPALASINGAM T, VAN BERKEL L, BLOEM P, et al. Relational Graph Convolutional Networks: a Closer Look [J]. PeerJ Computer Science. PeerJ Publishing, 2022, 8: e1073.
[16] SCHLICHTKRULL M, KIPF T, BLOEM P, et al. Modeling Relational Data with Graph Convolutional Networks [C]// Proceedings of the 15th European Semantic Web Conference. Cham: Springer, 2018: 593-607.
[17] SCHMIDHUBER J. Deep Learning in Neural Networks: An Overview [J]. Neural Networks. Elsevier, 2015, 61: 85-117.
[18] YAO Y. adore-ng [EB/OL]. (2015-12-30) [2023-04-03]. https://github.com/yaoyumeng/adore-ng.
[19] HAN J. Wukong: A LKM Rootkit for Linux Kernel 2.6.x, 3.x and 4.x [EB/OL]. (2016-04-07) [2023-04-03]. https://github.com/hanj4096/wukong.
[20] IPSecs. Kbeast-v1 [EB/OL]. (2012-01-01) [2023-04-03]. http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/.
[21] Chokepoint. JynxKit2 [EB/OL]. (2012-12-15) [2023-04-03]. https://github.com/chokepoint/Jynx2.
[22] En14c. LilyOfTheValley [EB/OL]. (2017-12-25) [2023-04-03]. https://github.com/En14c/LilyOfTheValley.
[23] SONG L, YIN H, LIU C. DeepMem [EB/OL]. (2019-07-06) [2023-04-03]. https://github.com/bitsecurerlab/DeepMem.
[24] 昌武洋, 付雄, 王俊昌. 基于 eBPF 与 LSTM 的 DDoS 攻击检测系统[J]. 重庆工商大学学报(自然科学版), 2023, 40(2): 36-43.

CHEN Liang, born in 1998, master, engineer. His main research interests include software security and memory forensics.

SUN Cong, born in 1982, Ph.D, professor, Ph.D supervisor, is a member of CCF(No. 28286M). His main research interests include software security, program analysis, and high-confidence software.

(责任编辑:何杨)