DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis

Wei Song
University of California, Riverside
wsong008@ucr.edu

Heng Yin
University of California, Riverside
heng@cs.ucr.edu

Chang Liu
University of California, Berkeley
liuchang@eecs.berkeley.edu

Dawn Song
University of California, Berkeley
dawnsong@cs.berkeley.edu

ABSTRACT

Kernel data structure detection is an important task in memory forensics that aims at identifying semantically important kernel data structures from raw memory dumps. It is primarily used to collect evidence of malicious or criminal behaviors. Existing approaches have several limitations: 1) list-traversal approaches are vulnerable to DKOM attacks, 2) robust signature-based approaches are not scalable or efficient, because it needs to search the entire memory snapshot for one kind of objects using one signature, and 3) both list-traversal and signature-based approaches all heavily rely on domain knowledge of operating system. Based on the limitations, we propose DeepMem, a graph-based deep learning approach to automatically generate abstract representations for kernel objects, with which we could recognize the objects from raw memory dumps in a fast and robust way. Specifically, we implement 1) a novel memory graph model that reconstructs the content and topology information of memory dumps, 2) a graph neural network architecture to embed the nodes in the memory graph, and 3) an object detection method that cross-validates the evidence collected from different parts of objects. Experiments show that DeepMem achieves high precision and recall rate in identify kernel objects from raw memory dumps. Also, the detection strategy is fast and scalable by using the intermediate memory graph representation. Moreover, DeepMem is robust against attack scenarios, like pool tag manipulation and DKOM process hiding.

CCS CONCEPTS

• **Applied computing** → **System forensics**; • **Computing methodologies** → *Neural networks*; • **Security and privacy** → Operating systems security;

KEYWORDS

Memory Forensics; Direct Kernel Object Manipulation; Deep Learning

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243813

ACM Reference Format:
Wei Song, Heng Yin, Chang Liu, and Dawn Song. 2018. DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243734.3243813

1 INTRODUCTION

Memory forensic analysis [6] extracts live digital evidence of attack footprints from a memory snapshot (or dump) of a running system. For instance, by identifying `_EPROCESS` objects in a Windows memory dump, analysts can figure out what processes are running on the target operating system. Memory forensic analysis is advantageous over the traditional disk-based forensics because although stealth attacks can erase their footprints on disk, they would have to appear in memory to run.

In previous works, researchers have explored memory forensics in OS kernels [7, 37], user-level applications [2, 42], as well as mobile devices [26, 27]. In this work, we focus on detecting objects in the kernel space. This problem is further complicated by kernel-mode attacks [16].

Generally speaking, the existing memory forensic tools fall into two categories: signature scanning and data structure traversal, all based on certain rules (or constraints), either on values, points-to relations, or both. Signature scanning tools (e.g., `psscan`) in Volatility [37] rely only on value constraints on certain fields to identify memory objects in the OS kernel, whereas SigGraph [20] relies on points-to relations as constraints to scan kernel objects. Data structure traversal tools (e.g., `pslist`) in Volatility and KOP [5] start from a root object in a known location, traverse its pointers to discover more objects, and further traverse pointers in the discovered objects to reach more objects.

However, there exist several intertwining challenges in the existing rule-based memory forensic analysis:

(1) **Expert knowledge needed**. To create signatures or traversing rules, one needs to have expert knowledge on the related data structures. For a closed-source operating system (like Windows), obtaining such knowledge is nontrivial if not impossible.
(2) **Lack of robustness.** Attackers may directly manipulate data and pointer values in kernel objects to evade detection, which is known as DKOM (Direct Kernel Object Manipulation) attacks [9]. In this adversarial setting, it becomes even more challenging to create signatures and traversing rules that cannot

be easily violated by malicious manipulations, system updates, and random noise.

(3) **Low efficiency.** High efficiency is often contradictory to high robustness. For example, an efficient signature scan tool (like `psscan`) simply skips large memory regions that are unlikely to have the relevant objects (like `_EPROCESS`) and relies on simple but easily tamperable string constants as constraints. In contrast, a robust signature scan tool would have to scan every single byte and rely on more sophisticated constraints (such as value ranges, points-to relations) that are more computation-intensive to check.

In this work, we are inspired by the successful adoption of deep learning in many problem domains (such as computer vision, voice, text, and social networks). We treat this memory object recognition problem as a deep learning problem. Instead of specifying deterministic rules for a signature scan and data structure traversal, we aim to learn a deep neural network model to automatically recognize memory objects from raw memory dumps. Since the model is trained in an end-to-end manner, no expert knowledge is required. The learned deep neural network model is also more robust than rule-based search schemes because it comprehensively evaluates all memory bytes and thus can tolerate perturbations to some extent. A deep neural network model also excels in efficiency, as vector and matrix computations can be largely parallelized in modern GPUs.

More specifically, in order to take into account adjacency relations between data fields within an object as well as points-to relations between two objects, we choose to build a graph neural network model [30], in which each node represents a segment of contiguous data values between two pointers, and each directed edge represents an adjacency relation or a points-to relation between two nodes. We then conduct supervised learning on this model: we collect a large number of diverse memory dumps, and label the objects in them using existing memory forensic tools like Volatility, and train the classification model using this labeled dataset.

We implement a prototype called DeepMem and conduct the extensive evaluation with respect to accuracy, efficiency, and robustness. Experimental results show that it achieves high precision and recall rate at above 99.5% for important kernel objects, like `_EProcess` and `_EThread`. For efficiency, it scans a memory dump of 1GB in size only once to build the memory graph in about 80 seconds. Then, for each type of object, the detection time is about 13 seconds per type on a moderate desktop computer (Core i7-6700, 16GB RAM, and no GPU). Moreover, in the attack scenarios, like pool tag manipulation and DKOM process hiding, signature-based memory forensics tool (e.g. Volatility), fail to correctly report kernel objects while DeepMem can tolerate those attacks.

In summary, the contributions of this paper are as follows:

- **A graph representation of raw memory.** We devise a graph representation for a sequence of bytes, taking into account both adjacency and points-to relations, to better model the contextual information in memory dumps.
- **A graph neural network architecture.** We propose a graph-based deep learning architecture with two jointly-trained networks: embedding network and classifier network. This deep neural network architecture captures both internal patterns of memory bytes as well as points-to structures in the memory graph and infers node properties in the graph.
- **A weighted voting scheme for object detection.** We propose a weighted voting scheme for object detection, which summarizes and cross-validates the evidence collected from multiple parts of an object to infer its location and type.

The remaining sections are structured as follows. Section 2 provides a background of memory object detection. Section 3 gives an overview of the DeepMem, followed by design details of each component. Section 4 presents implementation details and evaluation results. Section 5 discusses the limitations of our current design and implementation and sheds lights on future work. Section 6 surveys additional related work. In the end, Section 7 concludes the paper.

2 MEMORY OBJECT DETECTION

In this section, we first give a formal problem statement for memory object detection, and then describe the existing techniques and discuss their limitations. In the end, we share our insights.

2.1 Problem Statement

If we treat a memory dump as a sequence of bytes, an object in this dump are treated as a sub-sequence in this memory dump. Naturally, we can define the object detection problem as a subsequence labeling problem in a large sequence.

Our goal is to search and identify kernel objects in raw memory images dumped from running operating systems. Let $C = \{c_1, c_2, ...\}$ be the set of kernel data structure types in operating system. Given a raw memory dump as input, the output is defined as a set of kernel objects $O = \{o_1, o_2, ...\}$, where each object in the set is denoted as a pair $o_i = (addr_i, c_i), c_i \in C$. Here, $addr_i$ is the address of the first byte of the object in kernel space, and c_i is the type of the kernel object.

We would like to achieve the following goals:

- **No reliance on source code.** Unlike MAS [7] and KOP [5], which rely on the kernel source code to compute a complete kernel object graph, we do not assume the access to such information. Instead, we resort to learn from real memory dumps.
- **Automatic feature selection.** We do not rely on human experts to define signatures or traversing rules for various kernel objects. We aim to automatically learn a detection model in an end-to-end manner.
- **High robustness.** Our method should tolerate content and pointer manipulation of attackers in DKOM attacks.
- **High efficiency.** We would like to design a scanning approach to examine every byte in the memory, and at the same time, achieve high efficiency and scalability.

2.2 Existing Techniques

There are two approaches to utilize the knowledge of data structures for memory analysis.

The first one is data structure traversal. We can first identify a root object based on the data structure definition and then follows the pointers defined in this object to find more objects. In particular, Volatility [37], a well-known memory forensic tool, provides a set

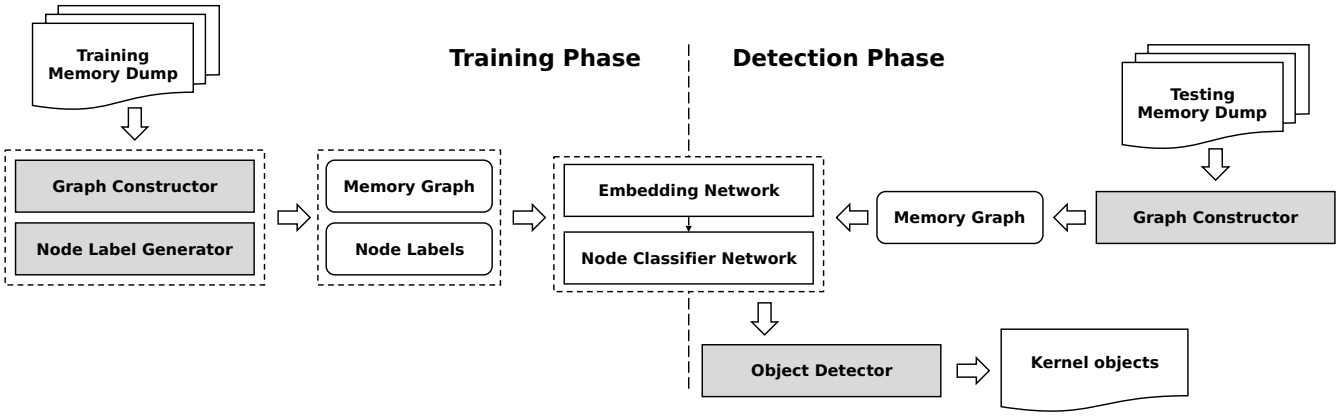


Figure 1: The overview of the DeepMem architecture

of tools for listing running processes, modules, threads, network connections, by traversing the relevant data structures. Since data structure definitions in C/C++ are often vague and incomplete (due to the presence of generic pointers), the completeness of this approach is affected. To address this problem, KOP [5] and MAS [7] perform points-to analysis on the C/C++ source code to resolve the concrete types for the generic pointers, and thus produce complete data structure definitions. This approach is efficient (as we can quickly find more objects by just following pointers), but not robust because attackers may modify the pointers to hide important objects, known as Direct Kernel Object Manipulation (DKOM) attacks.

The second approach is signature scan. We can scan the entire memory snapshot for objects that satisfy a unique pattern (called signature). Volatility [37] provides a set of scan tools as well to scan for processes, modules, etc. To improve search accuracy, SigGraph [20] automatically constructs graph-like signatures by taking into account points-to relations in data structure definitions, at the price of even lower search efficiency. In general, the signature scan is more resilient against DKOM attacks, because it does not depend so much on pointers. However, it is very inefficient and not scalable, because it has to search the entire memory snapshot for one kind of objects using one signature. To further improve the robustness of signatures, Dolan-Gavitt et al. [11] proposed to perform fuzz testing to mutate each data structure field and eliminate from the signature the constraints that can be easily violated by attackers. However, this will likely lead to the increase of false positives.

Both data structure traversal and signature scan require precise knowledge of data structures and also heavily depend on specific versions of the software or the operating system, because data structures change from one version to another. Therefore, to use these tools, a data profile must be extracted from each unique operating system version, which is clearly not convenient or scalable. To address this problem, researchers proposed to reuse the code already existed in the memory snapshot to interpret the memory snapshot itself [10, 13, 29]. These techniques avoid creating data profiles and implementing traversal algorithms, but they still heavily rely on the knowledge of specific operating systems to understand what code to reuse and how to reuse the code. Moreover, this approach is still subject to DKOM attacks. In terms of efficiency, code reuse is better than signature scan, but worse than data structure traversal.

2.3 Our Insight

We believe that the bottleneck for these memory analysis approaches is the rule-based search scheme. They search and traverse memory objects based on pre-defined rules. The rules can be hard to construct in the first place, and moreover, the rules cannot easily adapt to an unknown operating system and a new version and tolerate malicious attackers that attempt to deliberately violate these rules. To address these limitations, a "learning" ability becomes essential. A new memory analysis approach should automatically learn the intrinsic features of an object that are stable across operating system versions and resilient against malicious modifications, and at the same time is able to detect these objects in a scalable manner. In this work, we resort to deep learning to tackle this problem.

3 DESIGN OF DEEPMEM

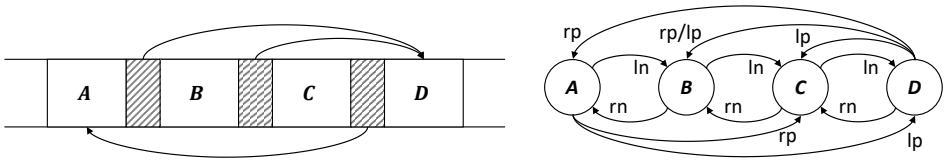
In this section, we first present an overview of DeepMem, and then delve into three important components respectively.

3.1 Overview

Figure 1 illustrates the overview of DeepMem. Generally speaking, we divide DeepMem into two separate stages: training and detection.

3.1.1 Training Stage. In this stage, DeepMem automatically learns the representation of kernel objects from raw bytes. First, memory dumps are fed into a graph constructor to generate a graph for each memory dump (which is called "memory graph"), where each node is a segment between two pointers, and each edge represents either an adjacency relation or a points-to relation between two nodes.

Second, a node label generator will assign a label for each node in the memory graph. We can use any existing tools (such as Volatility [37], or dynamic binary analysis tool DECAF [15]) for this purpose. This seems a little contradictory: we rely on an existing analysis tool to build a new analysis tool. This is reasonable because the existing tool only serves as an offline training purpose, so it does not need to be efficient and robust. It only needs to have reasonable



(a) Raw Memory (b) Memory Graph

Figure 2: Generate a memory graph from raw memory

accuracy in terms of labeling. After training, our detection model is expected to achieve good efficiency, robustness, and accuracy simultaneously.

Third, a memory graph is fed into a graph neural network architecture. By propagating information from neighboring nodes after several iterations, this graph neural network carries a latent numeric vector (called embedding) for each node in the memory graph.

Finally, all nodes' embedding vectors will go through a neural network classifier to get the predicted labels. The predicted labels will be compared with the expected labels to compute the loss of the classifier and update the weights of our neural network.

3.1.2 Detection Stage. this stage, DeepMem accepts an unlabeled raw memory dump and detects kernel objects inside it. First, it follows the same procedure to generate a memory graph for this memory dump. Second, the memory graph is fed into the Graph Neural Network (GNN) model obtained from the training stage to generate embeddings of all the nodes and then predict node labels using the neural network classifier. At last, DeepMem performs an object detection process. This is because the labels predicted from the last step are for segments, and an object may consist of one or several segments. Therefore, the object detection process takes segment labels as input and uses a voting mechanism to detect objects, for which most of their segment labels agree upon the same object label.

In the remainder of this section, we will discuss the definition of memory graph and its construction in Section 3.2, the graph neural network model for computing memory segments' embeddings as well as the segment classification network in Section 3.3, and object detection scheme in Section 3.4.

3.2 Memory Graph

A memory graph is a directed graph $G = (N, E_{ln}, E_{rn}, E_{lp}, E_{rp})$, where:

- N is a node set, and each $n \in N$ represents a segment of contiguous memory bytes between two pointer fields.
- E_{ln} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is left neighbor of n_j.
- E_{rn} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is right neighbor of n_j.
- E_{lp} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is pointed by a pointer on the left boundary of n_j.
- E_{rp} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is pointed by a pointer on the right boundary of n_j.

In other words, a memory graph is a directed graph with four sets of edges, which capture both the adjacency and points-to relations of memory segments, on both left-hand-side and right-hand-side of each segment.

Figure 2 illustrates an example of how to construct a memory graph from raw memory. Figure 2(a) shows a part of raw memory, in which three pointer fields split this part of memory into four segments: A, B, C, and D, each of which may have one or more contiguous memory bytes. As a result, A, B, C, and D become vertices in the corresponding memory graph. These vertices are connected by four kinds of edges. For instance, since A is the left neighbor of B, we have $A \xrightarrow{\text{ln}} B$. Conversely, since B is the right neighbor of A, we have $B \xrightarrow{\text{rn}} A$. Moreover, since the pointer field left to C points to D, and the pointer field right to C points to A, we then have $D \xrightarrow{\text{lp}} C$ and $A \xrightarrow{\text{rp}} C$. Note that these two edges are reverse to the actual points-to directions. This is because an edge in the memory graph represents an information flow. For instance, the pointer field left to C points to D, which means determining D's label can help label C. Therefore, from the information flow point of view, there is an edge from D to C.

A special case is that there are multiple consecutive pointers. Assume there are two consecutive pointers between C and D, pointing to A and B respectively, we then create four edges $A \xrightarrow{\text{rp}} C$, $B \xrightarrow{\text{rp}} C$, $A \xrightarrow{\text{lp}} D$ and $B \xrightarrow{\text{lp}} D$.

A careful reader might suggest adding edges for the points-to directions as well. For instance, the pointer field left to C points to D, and it might make sense to have $C \rightarrow D$, because identifying C also helps to identify D. We choose not to do so, because an adversary can easily create a pointer in an arbitrary address outside of a kernel object and make it point to the object, then the topology of the object in memory graph is changed if we add edges for point-to directions. This will adversely affect the detection. On the other hand, compared to the above case, it is more difficult to create a fake pointer or manipulate an existing pointer within a legitimate object that he/she tries to hide, without causing system crashes or other issues.

3.3 Graph Neural Network Model

The GNN (Graph Neural Network) model will accept the memory graph generated in Section 3.2 as input, and then output the labels of all nodes in the graph. The goal of GNN model is to first extract

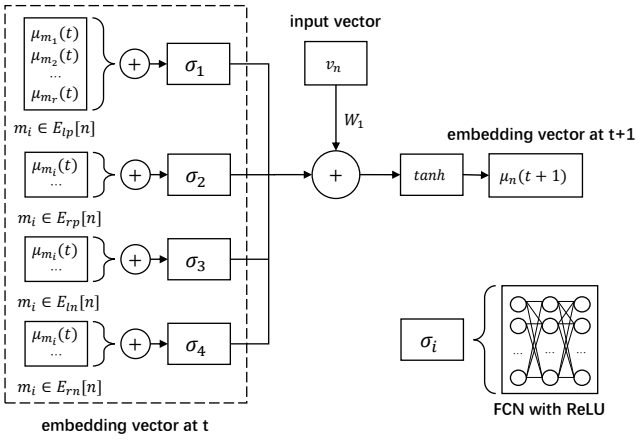


Figure 3: Node embedding computation in each iteration t. Information flows through E_{lp}, E_{rp}, E_{nl}, E_{nr} edges. Embedding vector $\boldsymbol{\mu}_n(t+1)$ gets updated by input vector v_n and its neighbors' embedding vectors at t.

a low-dimensional internal representation of nodes from raw bytes of a memory dump, and then infer the properties of nodes. As such, the GNN model should consist of two consecutive subtasks: a representation learning task and an inference task.

We represent the GNN model as $\mathcal{F}$. It consists of two jointly-trained subnetworks. The first subnetwork is an embedding network which is responsible for node representation abstraction. We denote it as ϕ_{w_1}. The second subnetwork is a classifier network, which is responsible for node label inference. We denote it as ψ_{w_2}. The formal definition of $\mathcal{F}$ is defined as follows.

$$\mathcal{F} = \psi_{w_2}(\phi_{w_1}(\cdot)) \tag{1}$$

The input of the embedding network ϕ_{w_1} is a vector representation of a node, denoted as $\boldsymbol{v}_n$, and the output is embedding vector, denoted as $\boldsymbol{\mu}_n$. The classifier network ψ_{w_2} takes the output of the embedding network as input, and then output the node label, denoted as $\boldsymbol{y}_n$.

More specifically, let $\boldsymbol{v}_n$ be a d-dimensional vector of node n derived from its actual memory content, then the embedding vector $\boldsymbol{\mu}_n$ is computed as follows:

$$\boldsymbol{\mu}_n = \phi_{w_1}(\boldsymbol{v}_n, \boldsymbol{\mu}_{E_{ln}[n]}, \boldsymbol{\mu}_{E_{rn}[n]}, \boldsymbol{\mu}_{E_{lp}[n]}, \boldsymbol{\mu}_{E_{rp}[n]}) \tag{2}$$

In other words, each node's embedding is computed from its actual content and the embeddings of its four kinds of neighboring nodes. We use a simple method to derive a d-dimensional vector for each node: we treat each dimension as one memory byte. If this memory segment is longer than d bytes, we truncate it and only keep d bytes; if it is shorter than d bytes, we fill the remaining bytes with 0.

Then the output vector $\boldsymbol{y}_n$ is computed as follows.

$$\boldsymbol{y}_n = \psi_{w_2}(\boldsymbol{\mu}_n) \tag{3}$$

In the following paragraphs, we will describe how embedding network and classifier network are defined and how they work.

3.3.1 Embedding Network. For each node n in the memory graph G, the embedding network ϕ_{w_1} integrates input vector v_n and the topological information from its neighbors, both adjacent neighbors and point-to neighbors, into a single embedding vector $\boldsymbol{\mu}_n$.

Inspired by Scarselli et al. [30], we implement the embedding vector as a state vector that gradually absorbs information propagated from multiple sources over time. To add a time variable into embedding vector computation, we transform Equation (2) into Equation (4). The total iterations needed to calculate the embedding vector is denoted as T. The embedding vector of time $t+1$ depends on the neighbor embedding vectors at time t, as shown in Figure 3.

$$\begin{aligned}\boldsymbol{\mu}_n(t+1) = \phi_{w_1}(\boldsymbol{v}_n, \boldsymbol{\mu}_{E_{ln}[n]}(t), \boldsymbol{\mu}_{E_{rn}[n]}(t),\\ \boldsymbol{\mu}_{E_{lp}[n]}(t), \boldsymbol{\mu}_{E_{rp}[n]}(t))\end{aligned} \tag{4}$$

For each node n, the embedding network collects the information about neighbor nodes in a BFS (Breadth First Search) fashion. In each iteration, it traverses one layer of neighbor nodes and integrates the neighbors' states into the state vector $\boldsymbol{\mu}_n$ of node n. We name the neighbors expanded in the first layer as 1-hop neighbors, in the same way, the neighbors expanded in the k-th layer as k-hop neighbors. In each layer expansion, we collect information from four types of neighbors, which are left neighbor, right neighbor, left pointer neighbor and right pointer neighbor. The more iterations we run, the information of farther neighbors are collected into embedding vector $\boldsymbol{\mu}_n$. At time $t = T$, $\mu_n(t)$ stores the information of the node sequence n itself and the information of neighbor nodes within T hops.

We implement embedding vector $\boldsymbol{\mu}_n$ as Equation (5).

$$\boldsymbol{\mu}_n(t+1) = tanh(W_1 \cdot \boldsymbol{v}_n + \beta(n,t)) \tag{5}$$

$$\begin{aligned}\beta(n,t) =& \sigma_1(\sum_{m \in E_{pt}[n]} \boldsymbol{\mu}_m(t)) + \sigma_2(\sum_{m \in E_{rn}[n]} \boldsymbol{\mu}_m(t)) +\\ & \sigma_3(\sum_{m \in E_{lp}[n]} \boldsymbol{\mu}_m(t)) + \sigma_4(\sum_{m \in E_{rp}[n]} \boldsymbol{\mu}_m(t))\end{aligned} \tag{6}$$

The weight matrix W_1 is the weight parameters of the node content, which is a matrix of shape $|\mu| \times d$. Neighbor state weight parameters are a set of weight matrices in multiple layered neural networks. Note that there are four separate sets of weight matrices for σ_1, and σ_2, and σ_3, and σ_4, such that the embeddings of different kinds of neighbors are propagated differently. The architecture of each σ network is a feed-forward neural network, each layer is a fully connected layer with ReLU activation function. The pseudo code of embedding network is shown in Algorithm 1.

All of the mentioned weight parameters of embedding network are learned using supervised learning on a labeled training dataset. Since the weights are learned jointly with the weights in the classifier network, we will leave the training details after introducing the classifier network in the section below. The embedding vector obtained in this section is just an intermediate representation of the whole supervised training. To perform an end-to-end training from raw bytes to labels, we need the classifier network to generate the final node label for training.

Algorithm 1: Information Propagation Algorithm of Embedding Network ϕ_{w_1}

Input : Memory Graph $G = (N, E_{ln}, E_{rn}, E_{lp}, E_{rp})$, iteration time T

Output : Graph Embedding $\boldsymbol{\mu}_n$ for all $n \in N$

Initialize $\boldsymbol{\mu}_n(0) = 0$, for each $n \in N$
for $t = 1$ *to* T **do**
  for $n \in N$ **do**
    $\beta = \sigma_1(\sum_{m \in E_{rn}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_2(\sum_{m \in E_{ln}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_3(\sum_{m \in E_{lp}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_4(\sum_{m \in E_{rp}[n]} \boldsymbol{\mu}_m(t-1))$
    $\boldsymbol{\mu}_n(t) = tanh(W_1 \cdot \boldsymbol{v}_n + \beta)$
  end
end

3.3.2 Classifier Network. Let l be a node label, and L be the set of all node labels. Node classifier network is used to map embedding vector to a node label: $\psi_{w_2} : \boldsymbol{\mu}_n \rightarrow l$, where $n \in N, l \in L$.

In order to facilitate object detection that will be discussed in Section 3.4, we choose to label each node as a 3-tuple of the object type, offset and length. For example, a node with label T_16_24 means the node is part of a _ETHREAD object and it is located at offset 16 from the beginning of the _ETHREAD object, the length of it is 24 bytes. As illustrated in Figure 4, three nodes are labeled as T_16_24, T_52_12 and T_84_28. These labels all agree upon a single fact that a _ETHREAD object is located at the same address. Similar labeling methods are adopted in the linguistics domain to solve word segmentation tasks [40, 41]. In particular, they label the characters at the start, in-between and at the end of a word, in order to split words from streams of free texts.

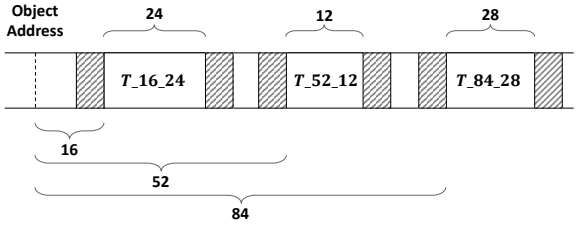


Figure 4: Node Labeling of a _ETHREAD Object

An object type may have many node labels. However, some rare and invariant node labels have low occurrences in type c. To get a robust model, we should not fit these outliers node labels. Hence, we just keep the node labels with high frequency in type c, denoted as key node label set $L(c)$. The node labeling method is described in detail in experiment evaluation Section 4.2.

With node labels of each object type, we then build a multi-class classifier to classify the nodes into one of the labels in that object type. For example, there will be a _ETHREAD classifier, a _EPROCESS classifier, etc. The node classifier takes an embedding vector $\boldsymbol{\mu}_n$ as input and produces a predicted node label as output. To implement the classifier, we choose to use FCN (Fully Connected Network) model that has multi-layered hidden neurons with ReLU activation functions, following by a softmax layer as the last layer.

After introducing the embedding network ϕ_{w_1} and the classifier network ψ_{w_2}, we will show how to train them together. During training, training samples are fed into the embedding network for contextual information collection. After propagating several iterations, the final embedding vectors are fed into the classifier network to generate the predicted output labels. To train the weights in the GNN model, we compute the cross-entropy loss between the predicted label and annotated label, and update weights in the process of minimizing the loss.

We adopt the BP (Back Propagation) [31] strategy to pass the loss error from output layer back to previous layers to update the weights along the way. In the next loop of training, the classification is performed using newly-updated weights. After several training loops, the loss will stabilize to a small value and the model is fully trained. Specifically, we use Adam (Adaptive Moment Estimation) [17] algorithm, a specific implementation of BP strategy, as the weight parameter optimizer of the GNN deep model.

Formally, let training dataset $D = \{d_1, d_2, ...\}$ be a set of node samples, where each sample $d_i = (\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)})$ is a pair of node vector and associated node label vector. The optimization goal is to compute the solution to Equation (7). $\mathcal{L}$ is the cross-entropy loss function that estimates the differences between classifier outputs and annotated labels. The parameters of embedding network w_1(including weights of $W_1, \sigma_1, \sigma_2, \sigma_3, \sigma_4$) and parameters of classifier network w_2 are updated and optimized in training.

$$\arg\min_{w_1, w_2} \sum_{i=1}^{|D|} \mathcal{L}(\boldsymbol{y}^{(i)}, \mathcal{F}(\boldsymbol{v}^{(i)}) \tag{7}$$

3.4 Object Detection

The basic idea behind object detection is that if several nodes indicate that there exists an object of certain type c at the same address s in the memory dump, we can conclude with a high confidence that we have detected an object of type c at that address, $c \in C$. Thus, a node label can be considered as a voter that votes for the presence of an object. For example, a node with a T_16_24 label means the node votes for the address, 16 bytes before the node address, to be the address of a _ETHREAD object. Each node in the memory indicates the presence of an object. Thus with all the node labels, we can generate a set of candidate object addresses $S = \{s_1, s_2, ...\}$ and corresponding voters for each address.

Next, we need to determine whether an address $s \in S$ is indeed a start address of an object. Ideally, if all the key nodes of type c vote for s to be an object of type c, for example T_16_24, T_52_12, T_84_28... all suggest the presence of an _ETHREAD at the same address s, we can confidently report a _ETHREAD object is detected at s. It is also likely that only a fraction of the key node labels votes for address s, then our confidence to report address s will be lower. We use $L(s, c)$ to denoted the voter set, which is all the key node labels of type c that vote for address s.

Specifically, we design a weighted voting mechanism. It gives different node labels (or in other words voters) different vote weights.

Since the voter with higher frequency in a certain object type better indicates the presence of the objects of that type, and thus is assigned with a larger weight. The weights are calculated from a large real-world labeled dataset.

Finally, we introduce the prediction function $f(s,c)$ in Equation (8). It measures the difference between the prediction confidence and a pre-defined threshold δ. When the value of $f(s,c)$ exceeds the threshold, we draw a conclusion that an object with type c is detected at address s.

$$f(s,c) = \begin{cases} 1, & \sum_{l_i \in L(s,c)} \frac{\rho(c,l_i)}{\rho(c)} + \gamma(s,c)) > \delta \\ 0, & otherwise \end{cases} \tag{8}$$

Here, ρ is a counting function, $\rho(c)$ counts the number of objects of type c in the dataset, and $\rho(c,l)$ counts the number of objects of type c that has node label l in the dataset, $l \in L(c)$. Then, we divide $\rho(c,l)$ by $\rho(c)$ to estimate the weights of node label l in predicting objects of type c , which is a decimal value in $(0,1]$. Since the weight values of voters range in $(0,1]$, it is possible that weighted combination of multiple small-weighted voters is less than that of a large-weighted single voter (e.g. weight sum of two small voters 0.4 + 0.3 < weight value of a single large voter 0.8). In fact, the evidence from multiple voters is more persuasive than a single voter with a large weight, because it is less likely that two different voters both make errors and vote for the same address of the same type in a large and arbitrary memory space. So, we add a function $\gamma(s,c)$ to reward the cross-validated addresses voted by multiple voters.

In the implementation, the threshold δ is determined using a searching method in the validation dataset. We run the experiment by tuning the value of threshold δ to get the one that yields the highest F-score [24], and set it as the default threshold. The reward function is devised as $\gamma(s,c) = |L(s,c)| - 1$.

4 EVALUATION

In this section, we first describe the experiment setup in Section 4.1. Then, we discuss the dataset collection and labeling approach in Section 4.2. Section 4.3 provides details about training. In the end, we present the evaluation results with respect to accuracy, robustness, and efficiency in Section 4.4, 4.5, and 4.6 respectively.

4.1 Experiment Setup

Our experiment uses two settings of configurations. 1) The training experiment is performed on a high-performance computing center with each worker node equipped with 32 cores Intel Haswell CPUs, 2 x NVIDIA Tesla K80 GPUs and 128 GB memory. 2) The detection experiment is performed on a moderate desktop computer with Core i7-6700, 16GB, no GPU. We use powerful GPUs on the computing center for training, which is a one-time effort. Once the model is trained, it is loaded on a desktop computer to conduct the kernel object detection.

The deep neural network models in DeepMem, like embedding network and classifier network, are all implemented using the open-source deep learning framework TensorFlow [1]. The remaining codes of data processing, statistics, plotting are programmed in Python.

4.2 Dataset

4.2.1 Memory Dumps Collection. While DeepMem can analyze any operating system versions in principle, it is limited by the object labeling tool used in training. In the evaluation, we choose to evaluate DeepMem on Windows 7 X86 SP1 rather than the latest Windows 10, mainly because the object labeling tool we used, Volatility [37], was unable to consistently parse Windows 10 images or memory dumps, but worked very stable for Windows 7 images.

To automatically collect a large number of diverse memory dumps for training and detection, we developed a tool with two functionalities: 1) simulating various random user actions, and 2) forcing the OS to randomly allocate objects in the memory space between consecutive memory dumps.

To simulate various user actions, the memory collecting tool first starts the guest Windows 7 SP1 virtual machine which is installed in the VirtualBox [36]. When the virtual machine is started, guest OS automatically starts 20 to 40 random actions, including starting programs from a pool of the most popular programs, opening websites from a pool of the most popular websites, and opening random PDF files, office documents, and picture files. Next, the memory collecting tool waits for 2 minutes and then dumps the memory of the guest system to a dump file. When the dump is saved to the hard disk of the host system, it restarts the virtual machine and repeats until we collect 400 memory dumps, each of which is 1GB in size.

To ensure kernel objects to be allocated at random locations, we enabled KASLR when generating our dataset and restarted the virtual machine after each dump. We found out that the address allocations of objects are different among different memory dumps. Only 1.32% `_EPROCESS` objects in a memory dump are located at the same virtual address of `_EPROCESS` objects in another dump. The ratio is 4.7% for `_ETHREAD`, 0.68% for `_FILE_OBJECT`, 15.9% for `_DRIVER_OBJECT`. The basic statistics of memory dumps and memory graphs are shown in Table 1.

Kernel Object Type	Mean Count	Std Dev
_EPROCESS	85	7.47
_ETHREAD	1,216	112.25
_FILE_OBJECT	3,639	918.06
_DRIVER_OBJECT	109	0.22
_LDR_DATA_TABLE_ENTRY	141	0.59
_CM_KEY_BODY	1,921	953.76
Memory Graph Statistics	**Mean Count**	**Std Dev**
Nodes	1,334,822	134,564.24
Edges	5,325,214	513,624.71

Table 1: Statistics of memory dumps and memory graphs.

4.2.2 Memory Graph Construction. To generate a memory graph, we first read and scan all available memory pages in the kernel virtual space of memory dumps. Then, we locate all the pointers in the pages by finding all fields whose values fall into the range of kernel virtual space. For each segment between two pointers, we create a node in the memory graph. For each node, we find its neighbor nodes in the memory dump according to the neighbor definitions in Section 3.2, and create an edge in the memory graph.

Kernel Object Types	Object Length	Avg. #TP	Avg. #FP	Avg. #FN	Precision%	Recall%	F-Score
_EPROCESS	704	82.834	0.017	0.303	99.979%	99.635%	0.99807
_ETHREAD	696	1211.476	5.514	0.7	99.547%	99.942%	0.99744
_DRIVER_OBJECT	168	108.938	0.255	0.024	99.766%	99.978%	0.99872
_FILE_OBJECT	128	3621.007	67.545	23.045	98.169%	99.368%	0.98765
_LDR_DATA_TABLE_ENTRY	120	139.093	0.0	2.4	100.0%	98.304%	0.99145
_CM_KEY_BODY	44	1979.207	94.621	0.414	95.437%	99.979%	0.97655

Table 2: Object Detection Results on Memory Image Dumps.

4.2.3 Node Labeling. The node labeling process takes four steps: 1) utilize Volatility to find out the offset and length information of 6 kernel object types (i.e. `_EPROCESS`, `_ETHREAD`, `_DRIVER_OBJECT`, `_FILE_OBJECT`, `_LDR_DATA_TABLE_ENTRY`, `_CM_KEY_BODY`) in memory dumps; 2) for each node in the memory graph, determine if it falls into the range of any kernel object, and if so, calculate the offset and length of that node in that kernel object and give the node a label; 3) select the top 20 most frequent node labels across all kernel objects of type c as key node label set $L(c)$ for type c; and 4) label the rest nodes in the memory graph as *none*.

4.2.4 Sample Balancing. Inside a large memory dump, kernel objects only take up a small portion of the memory space. Thus, the key nodes of kernel objects in the memory graph are very sparse. Also, the key nodes of a certain object type are not evenly distributed. To accelerate the training process and achieve better detection results, we need to balance samples in the training dataset.

The principle of balancing is to preserve the topologies of the key nodes in the memory graph after the balancing process. Specifically, 1) to reduce non-key nodes, we remove the nodes that are k-hops away from key nodes in memory graph (k is a predefined value), 2) to increase key nodes and balance between different node types, we duplicate the key nodes to the same amount, and also duplicate the edges between nodes in edge matrix. Since the embedding vector is calculated using inward edges only, such duplication does not create new neighbors for the original key nodes, so it does not affect the topology propagation of the original key nodes.

4.3 Training Details

We split the collected 400 memory dumps into 3 subsets. We randomly select 100 images as the training dataset, 10 images as the validation dataset and the remaining 290 images as the testing dataset. The validation dataset and testing dataset will not be used in the training phase, and this guarantees that the detection model never sees the testing set in the training phase.

In each training iteration, we randomly select an image from the training dataset for training. To determine whether the model is fully trained, we monitored the loss and accuracy on the validation dataset during the training process. When the loss reaches a relatively small and stable value, we deem the model as fully trained or it reaches its learning capacity. Dropout layers [35] are added to prevent the over-fitting problem. We set the keep probability to 0.8 in the training phase, and to 1 in the evaluation phase and testing phase.

By default, the experiments are all performed under the same parameter setting as described in Table 3.

Parameters	Value
Layers of σ	3
Layers of ψ	3
Optimizer	Adam Optimizer
Learning Rate	0.0001
Propagation Iteration T	3
Input Vector Dimension	64
Embedding Vector Dimension	64
keep_prob	0.8

Table 3: Default Parameters of Experiments.

4.4 Detection Accuracy

We measured the accuracy using a number of different metrics, including precision, recall, and F-score [24]. For each object type, precision calculates the correctly classified samples against all detected samples. Recall calculates the correctly classified samples against all labeled samples in this type. F-score is the harmonic mean of precision and recall.

Table 2 shows the detection results of various kernel object types on raw memory images by training for 13 hours. We can see from the result, the overall recall rate is satisfactory, ranging from 98.304% to 99.979%. Most large kernel objects ($\geq$ 120 bytes) have over 98% precision rate. Important kernel object types `_EPROCESS`, `_ETHREAD` both achieve over 99.6% recall rate, and over 99.5% precision rate. Also, we observed a tendency that larger objects achieve better recognition results. The reason is that for small objects, there are fewer nodes and pointers inside them. Then, the chance of obtaining stable key nodes is lower.

4.5 Robustness

For the evaluation of robustness, we performed three experiments. The first experiment is pool tag manipulation, with the aim to evaluate its impact on signature scanning tools and DeepMem. The second experiment is pointer manipulation, with the aim to evaluate if DeepMem is still effective in DKOM process hiding attacks. The third experiment is a general yet more destructive attack which is to randomly mutate arbitrary bytes in memory, with the aim to see whether our approach is resistant to various attack scenarios, and to what extent it can tolerate random mutations.

4.5.1 Pool Tag Manipulation. To perform pool tag manipulation, we change the 4 bytes pool tags [33] of each object to random values in the memory dump file. Using the manipulated dump, we then test the effectiveness of our approach and several Volatility plugins.

In our experiment, we randomly select 10 memory dumps as the testing set, and take scanning _FILE_OBJECT object as an example. As shown in Table 4, the filescan plugin of Volatility cannot correctly report _FILE_OBJECT objects. Its recall rate drops to a small value of 0.0082%. The reason is that filescan first needs to search for the pool tag of _FILE_OBJECT in the entire memory dump. As a result, most of the _FILE_OBJECT objects are not reported.

As a comparison, DeepMem works normally in evaluation results. It can achieve a recognition precision of 99.1% and recall of 99.05%. The reason is that DeepMem examines every byte of a memory dump to detect objects, rather than merely rely on pool tag constraints to locate objects. Hence, without valid pool tags, DeepMem can still detect objects in the memory dump. This indicates that approaches based on hard constraint matching are not robust. In contrast, our approach is based on soft features automatically learned from raw object bytes, which can capture a more robust representation of an object.

Method	Avg. #TP	Avg. #FP	Avg. #FN	Precision%	Recall%
filescan	0.3	0.0	3661.8	100%	0.0082%
DeepMem	3627.2	32.9	34.9	99.1%	99.05%

Table 4: Results of _FILE_OBJECT Pool Tag Manipulation

4.5.2 DKOM Process Hiding. This DKOM attack is to hide a malicious process by unlinking its connections to precedent and antecedent processes in a double linked list. In this case, list traversal related tools, like the pslist plugin in Volatility, will fail to discover the hidden process through this broken link list.

In our experiment, we randomly choose 20 memory dumps as a testing set, and then manipulated the value of the forward link field in each _EPROCESS object to random value. In Table 5, we can see that the Volatility plugin pslist fails to discover most _EPROCESS objects except the first one in each dump. Since the _EPROCESS list is broken by the manipulation, it cannot traverse through the double linked list to find other processes. In contrast, DeepMem can still find 99.77% _EPROCESS objects with 100% precision.

Method	Avg. #TP	Avg. #FP	Avg. #FN	Precision%	Recall%
pslist	1.05	0.0	85.7	100%	1.21%
DeepMem	86.55	0.0	0.2	100%	99.77%

Table 5: Results of DKOM Process Hiding Attacks

4.5.3 Random Mutation Attack. It is hard to simulate all kinds of DKOM attacks. Therefore, we take a simple approach to find out how much DeepMem can tolerate DKOM attacks: we gradually increase the number of bytes to be manipulated in random positions of kernel objects, including the pointer and non-pointer fields, and evaluate the precision and recall rate at different mutation levels. In Section 4.5.1 and Section 4.5.2, we have already demonstrated how DeepMem works on memory dumps with small changes. In this section, we will show how DeepMem perform when large bytes are changed.

Even if an attacker largely changes the contents and topologies in kernel objects of the operating system, DeepMem can be used in this scenario without retraining the detection model with the samples from that attack. We just need to lower the prediction threshold δ. However, in extreme case, if the threshold is set to a very small value, then most addresses in candidate address set S will be reported, causing many false positives and low precision. To guarantee a high precision while getting a recall as high as possible, it is better to report the objects cross-validated by at least two voters. This can be achieved by setting the threshold δ of prediction function $f(s, c)$ to 1 (If there are more than two voters, the reward function $\gamma(s, c) = |L(s, c)| - 1 \geq 1$, the prediction confidence > 1. See Equation (8)).

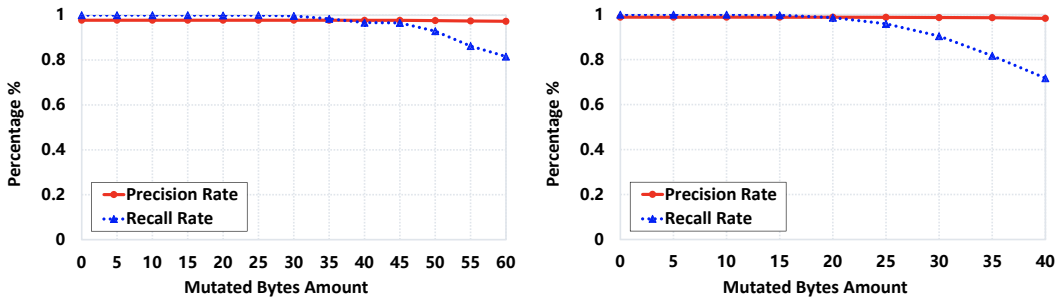
We evaluate the detection results by mutating different amount of bytes in objects for _EPROCESS and _ETHREAD objects, with threshold δ is set to 1. We can see from Figure 5, as the number of mutated bytes increases, the precision rate remains stable at around 97% - 98% with tiny perturbations. Recall rate curve stays at a high rate at first, then drops down as the number of mutated bytes further increases. Specifically, for _EPROCESS, it achieves over 97% precision rate at all mutation levels, and 100% recall rate before 20 bytes are changed. Our model can tolerate up to 50 bytes random mutation, without causing the precision and recall rate drop significantly. For _ETHREAD, our model can tolerate up to 30 bytes random mutation. We can see when we set the threshold δ to a low value 1, the precision rate does not drop significantly.

The causes of the high precision and recall rate are twofold. First, the neural network itself can inherently tolerate small mutations due to the robust features it learns from the training data. Second, even when deep model incorrectly predicts the labels of some nodes of an object, the remaining nodes can make cross-validation and collectively conclude the presence of an object. The recall rate indeed drops significantly with larger mutations. However, these larger mutations will likely cause system crashes or instability, and therefore might be rarely seen in real-world attacks.

4.6 Efficiency

To investigate the efficiency of DeepMem, we measure the time allocations in different phases. We consider three types of time consumption: GNN model training time T_t, memory graph construction time T_g and object detection time T_d. 1) The training time T_t measures the time from inputting raw labeled training dataset dumps to obtaining a fully trained model with a small and stable prediction loss. 2) The memory graph construction time T_g measures the time from inputting a raw memory dump to obtaining matrix representation of the memory graph. 3) The object detection time T_d measures the time from inputting a memory graph matrix to obtaining detected kernel object set of a certain object type. The experiment settings of training and detection are described in Section 4.1.

In the training phase, we utilize the GPU in the computing center to train the model because the major computation of training is matrix-based and GPU can accelerate the matrix computation. We train the model for 13 hours for one object type. After training, the model can be saved to disk and deployed in a desktop computer(with or without GPU). In our detection experiment, we copy the model to a moderate desktop computer without GPU. On average, it takes 79.7 seconds to construct the whole memory graph for one memory



(a) Random Mutation Attack of _EPROCESS Object (b) Random Mutation Attack of _ETHREAD Object

Figure 5: Random Mutation Attack

dump of 1GB size, and 12.73 seconds to recognize the objects of a certain type in it, as shown in Figure 6. This detection time can be accelerated by using GPU. In our computing center, the detection time can be reduced to about 7.7 seconds.

DeepMem is efficient for two reasons. First, it turns a memory dump into a graph structure denoted as large node matrices and edge matrices, which is especially suitable for fast GPU parallel computation. Second, since it converts the memory dump into an intermediate representation (memory graph), and performs the detection of various object types on this graph, there is no need to scan the raw memory multiple times to match the various set of signatures for different object types.

	Time Measurements	Mean	Std Dev
Training	Training T_t (per object type)	13 Hours	N/A
Detection	Graph Construction T_g (per dump)	79.7 Sec	6.64
	Object Detection T_d (per type)	12.73 Sec	1.24

Table 6: Time Consumption at Different Phases. Training is performed on the computing center. Detection is performed on a desktop computer. The setting is in Section 4.1

4.7 Understanding Node Embedding

We plot the embedding vectors of nodes using t-SNE visualization technique [22] in Figure 6. Each node embedding vector in multi-dimensional space is mapped as a point in two-dimensional space. We collect embedding vectors of different object types at the output layer of the embedding network before they are fed into the classifier network. Figure 6 shows the distribution of embedding vectors in 2D space, where different colors are used to denote different types of node labels. To clearly show plenty of embeddings of different types, we only plot the first 10 key nodes for each object type. We expect to observe that points of the same colors locate near each other, and different colors locate far from each other. From the figure, we can see that the visualized results meet that expectation. These embeddings can capture the intrinsic characteristics of nodes, and different types of nodes are well separated.

4.8 Impact of Hyperparameters

We plot ROC curves [14] of detection results to show the impact of the different hyperparameters of our model. We adjust three parameters: the propagation iteration times T, the embedding vector size, and the embedding depth of embedding network σ. ROC curve shows the trade-off between sensitivity (true positive rate) and specificity (false positive rate) of the object detector.

Figure 7(a) shows the performance of the _FILE_OJBECT detector by tuning the iteration parameter T of node embedding network ϕ. We can see that the ROC curve of $T = 3$ is nearest to the upper left corner, followed by the curves of $T = 2$ and $T = 1$. The trend demonstrates the importance of topological information propagation in object detection. With more information collected through propagation, the prediction ability of the object detector is further improved.

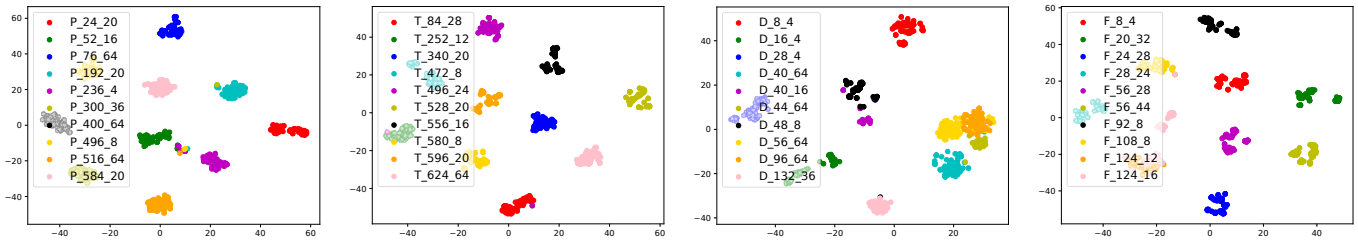
Figure 7(b) shows the performance of _FILE_OBJECT detector by tuning embedding vector size of node embedding network ϕ. In the figure, the ROC curve with larger embedding size is closer to the upper left corner. It shows that larger embedding vector size is more expressive and better approximate the data intrinsic characteristics. However, this is also a trade-off between learning ability and training time. In practical usages, for the same level of learning ability, a smaller embedding size is preferred for faster training and testing. The determination of such embedding size should be a combined consideration of the task complexity and training effort.

Figure 7(c) shows the performance of _FILE_OBJECT detector by tuning embedding layers depth of σ. In the figure, the ROC curve with more layers is closer to the upper left corner. It indicates that the learning ability of deeper neural network is stronger than shallower networks. Enlarging the number of layers and embedding size is a preferred solution for training complex object types.

5 DISCUSSION

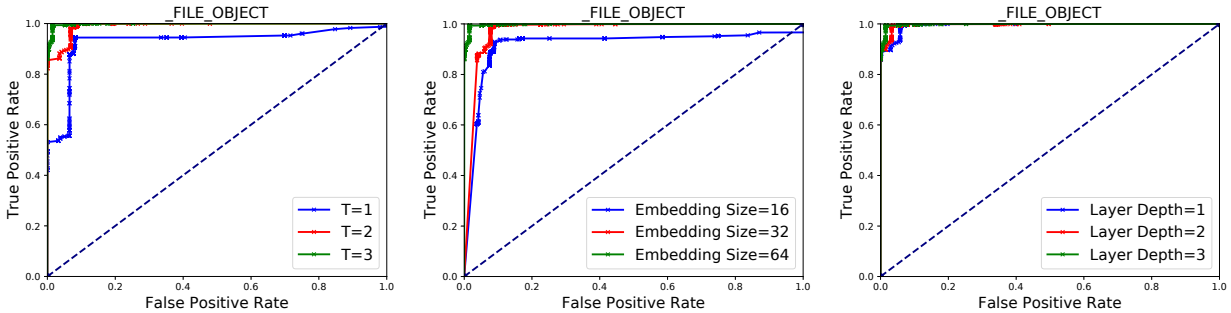
In this section, we discuss several limitations and potential issues related to DeepMem.

Small Objects. DeepMem may not perform well for small objects with few or no pointers, like many other pointer-based approaches [19]. Our approach model objects based on both content of objects and topological relations between objects. Small objects



(a) _EPROCESS (b) _ETHREAD (c) _DRIVER_OBJECT (d) _FILE_OBJECT

Figure 6: Node Embedding Visualization using t-SNE



(a) ROC versus Iterations T (b) ROC versus Embedding Vector Size (c) ROC versus Layer Depth of σ

Figure 7: ROC Curves by Tuning Parameters

lacking pointers are not informative enough and also have weak or no relations with other nodes in the memory. Thus very little information could be gathered from others nodes to make inference on the objects. Fortunately, important kernel objects like _EPROCESS, _ETHREAD and _DRIVER_OBJECT are long enough for our approach to achieve over 99.6% recall and over 99.5% precision rate, which is sufficient for general memory forensic purposes.

Data Diversity and Validity. To generate diverse dumps, we try to simulate random user actions and allocate kernel objects in random positions in the memory, as described in the evaluation section. Even with these efforts, our dataset may not be diverse enough. To make it more diverse, researchers can use different physical machines, load different drivers, etc. Nevertheless, our evaluation on the dataset at least demonstrates the feasibility of DeepMem in a homogeneous environment (e.g., an enterprise network in which all computers have the same configuration and in a cloud environment where VMs are instantiated from the same base image). We use Volatility to label memory dumps as ground truth. According to the paper [25], Volatility achieves zero FPs and FNs for most of their plugins for non-malicious dumps. So our training set labeling should not be affected. Plus, we can use other solutions to label memory dumps as suggested in this paper, such as using DECAF [15].

Cross Operating System Versions. In the evaluation phase, we have already demonstrated the robustness of our approach in scenarios like pool tag attack, DKOM process hiding and random bytes mutation. It shows that our approach tolerates well for small changes and manipulations of the memory. This feature is useful in real-world applications. For example, our approach will adapt to systems changes across versions and patches. We leave this for future work.

6 RELATED WORK

Memory forensic analysis aims at exploring the semantic content of interests from volatile memory of different platforms and operating systems, such as Windows [5, 12], Linux [19, 20], Android [19, 26–28], etc. Among them, kernel object recognition is a fundamental task. Basically, the approaches can be classified into two categories according to memory search methods: one is list-traversal [5, 21] approach, the other is signature-based scanning [3, 4, 11, 20, 23, 32].

List traversal approaches usually start searching objects from the global root in the memory, then gradually expand the search scope and find more objects by traversing along the point-to directions of pointers. KOP [5] applies inter-procedural points-to analysis to compute all possible types for generic pointers, uses a pattern matching algorithm to resolve type ambiguities, and utilizes knowledge of kernel memory pool boundaries to recognize dynamic arrays.

Signature scanning approaches usually scan the memory image from the start to the end sequentially. During the scanning, it tests whether the observed memory subsequences match the designed object signatures, then decides the object type of the sequence. SigGraph [20] utilize point-to relations between different objects to generate non-isomorphic signatures for data structures in an OS

kernel. Dimsum [19] constructs boolean constraints from data structure definition and memory page contents to build graphical models so that it can recognize data structure instances in un-mappable memory. Then it performs probabilistic inference to generate results ranked with probabilities. An object is detected once it satisfies all the constraints. Dimsum has slightly higher false negative rate than Value-Invariant and SigGraph, but it has significantly less false positive rate than those two systems. Dimsum conducts probabilistic inference and constraint solving to infer the address of kernel object. To do so, it needs to create many boolean variables for each memory location, making the factor graph very large and very expensive to resolve. So Dimsum introduces a pre-processing phase to reduce the number of locations to test. It may not be robust if the attackers find a way to evade the pre-processing phase.

In comparison, DeepMem is fundamentally different. The key to the list-traversal approach is to find the special global root from which extra objects are traversed and expanded through pointers. DeepMem does not need to start scanning from the root. It is able to examine every segment in the memory then comprehensively evaluate these segments and connections between them to make a holistic inference decision. We use pointers only in topological information propagation computation, in this case, an unlinked pointer would not have a huge impact on the propagation while list-traverse will completely stop working if a link is broken. The key to the signature-based approach is to find accurate and robust signatures for each type of kernel object. It needs to face problems like generic pointer problem, constraints explosion etc. DeepMem learns the pointer and non-pointer constraints automatically instead of using hard signatures or expert-made constraints, and captures non-linear relations between nodes in the graph. It is more expressive than signature-based approaches, and thus more accurate and robust. Moreover, both list-traversal and signature scanning depend partially or fully on operating source code or data structure definitions. DeepMem does not need this domain knowledge and specifications.

We also leverage several deep learning techniques [18] in graph modelings, such as node embedding and node classification. We use a modified Graph Neural Network [30] to model nodes that preserve local content information and contextual topological information through information propagation. Other researchers also make use of contextual information in the graph to solve the graph embedding problem [8, 38, 39]. We use Fully-Connected Neural Networks to make inference in node properties [34]. What is common in these models is that they are able to achieve an end-to-end learning, where patterns of data are automatically explored without domain knowledge or human intervention.

7 CONCLUSION

In this paper, we proposed a graph-based kernel object detection approach DeepMem. By constructing a whole memory graph and collecting information through topological information propagation, we can scan the memory dumps and infer objects of various types in a fast and robust manner. DeepMem is advanced in that 1) it does not rely on the knowledge of operating system source code or kernel data structures, 2) it can automatically generate features of kernel objects from raw bytes in memory dump without manual expert analysis, 3) it utilizes deep neural network architectures for efficient parallel computation, and 4) it extracts robust features that are resistant to attacks like pool tag manipulation, DKOM process hiding.

The experimental result shows that it performs well in terms of accuracy, robustness, and efficiency. For accuracy, it reaches above 99.5% recall and precision rate for important kernel objects like `_EPROCESS` and `_ETHREAD`. For robustness, the recognition result stays stable in different attack scenarios, like manipulating pool tags, pointers, and even random byte mutations. For efficiency, we turn a memory dump into an intermediate memory graph representation, and then detect objects of different types using the graph. The detection time of each object type is about 13 seconds.

ACKNOWLEDGEMENT

We appreciate the anonymous reviewers for the valuable comments. We thank Zhenxiao Qi for collecting dataset, Abhishek Srivastava for providing technical suggestions on TensorFlow and Sri Shaila for proofreading. The research work is supported by National Science Foundation under Grant No. 1664315, 1719175, TWC-1409915. This work was supported in part by FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166). This work is also supported by Center for Long-Term Cybersecurity from UC Berkeley. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*.

[2] Cosimo Anglano. 2014. Forensic analysis of WhatsApp Messenger on Android smartphones. (2014).

[3] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2008. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference (ACSAC)*.

[4] Chris Betz. 2018. MemParser. https://sourceforge.net/p/memparser/wiki/Home/.

[5] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 555–565.

[6] Andrew Case and Golden G Richard III. 2016. Memory forensics: The path forward. (2016).

[7] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. 2012. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of USENIX Security Symposium*.

[8] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*. 2702–2711.

[9] DKOM 2018. FU rootkit. https://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[10] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[11] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 566–577.

[12] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. 2014. MACE: high-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 196–205.

[13] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic-Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy(Oakland'12)*. IEEE, 586–600.

[14] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. (1982).

[15] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*.

[16] Greg Hoglund and James Butler. 2006. Rootkits: subverting the Windows kernel. (2006).

[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. (2014).

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. (2015).

[19] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. NDSS*.

[20] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the Network and Distributed System Security Symposium*.

[21] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*.

[22] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. (2008).

[23] Nick L Petroni, Aaron Walters, Timothy Fraser, and William A Arbaugh. 2006. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. (2006).

[24] David Martin Powers. 2011. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. (2011).

[25] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. 2015. On the Trustworthiness of Memory Analysis-An Empirical Study from the Perspective of Binary Execution. (2015).

[26] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. GUITAR: Piecing together android app GUIs from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 120–132.

[27] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[28] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. 2016. Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images.. In *USENIX Security Symposium*.

[29] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2014. DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse.. In *USENIX Security Symposium*.

[30] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. (2009), 61–80.

[31] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. (2015).

[32] Andreas Schuster. 2006. Searching for processes and threads in Microsoft Windows memory dumps. (2006).

[33] Andreas Schuster. 2008. The impact of Microsoft Windows pool allocation strategies on memory forensics. In *Digital Investigation, Volume 5*.

[34] Alexander G Schwing and Raquel Urtasun. 2015. Fully connected deep structured networks. (2015).

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. (2014).

[36] VirtualBox 2018. VirtualBox. https://www.virtualbox.org/.

[37] Volatility 2018. Volatility: Memory Forencis System. https://www.volatilityfoundation.org/.

[38] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*.

[39] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[40] Yue Zhang and Stephen Clark. 2008. Joint word segmentation and POS tagging using a single perceptron. (2008).

[41] Hai Zhao, Chang-Ning Huang, and Mu Li. 2006. An improved Chinese word segmentation system with conditional random field. In *Proceedings of the Fifth SIGHAN Workshop on Chinese Language Processing*.

[42] Fan Zhou, Yitao Yang, Zhaokun Ding, and Guozi Sun. 2015. Dump and analysis of android volatile memory on wechat. In *Communications (ICC), 2015 IEEE International Conference on*.