# MACE: High-Coverage and Robust Memory Analysis For Commodity Operating Systems

Qian Feng Aravind Prakash Heng Yin
Syracuse University
{qifeng,arprakas,heyin}@syr.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

## Abstract

Memory forensic analysis collects evidence for digital crimes and malware attacks from the memory of a live system. It is increasingly valuable, especially in cloud computing. However, memory analysis on on commodity operating systems (such as Microsoft Windows) faces the following key challenges: (1) a partial knowledge of kernel data structures; (2) difficulty in handling ambiguous pointers; and (3) lack of robustness by relying on soft constraints that can be easily violated by kernel attacks. To address these challenges, we present MACE, a memory analysis system that can extract a more complete view of the kernel data structures for closed-source operating systems and significantly improve the robustness by only leveraging pointer constraints (which are hard to manipulate) and evaluating these constraint globally (to even tolerate certain amount of pointer attacks). We have evaluated MACE on 100 memory images for Windows XP SP3 and Windows 7 SP0. Overall, MACE can construct a kernel object graph from a memory image in just a few minutes, and achieves over 95% recall and over 96% precision. Our experiments on real-world rootkit samples and synthetic attacks further demonstrate that MACE outperforms other external memory analysis tools with respect to wider coverage and better robustness.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

## General Terms

Security

## Keywords

Memory Analysis, Random Surfer, Rootkit Detection

## 1. INTRODUCTION

Memory analysis has become increasingly valuable in digital crime investigation and malware analysis, as it extracts *live* digital evidence of attack footprints from the volatile memory state of a running system, which cannot be obtained from traditional hard disk based forensic analysis. Memory analysis is particularly beneficial for cloud computing security, because one can quickly scan a large number of virtual machine states to detect malicious activities, without installing security agents (which is inconvenient and can be easily subverted) inside the virtual machines. For example, a recent work proposed to detect rootkit infestation in homogeneous virtual machines in the cloud [3].

However, there exist several long-standing challenges in memory analysis especially for closed-source operating system (OS) such as Microsoft Windows.

**(1) Low coverage.** Without access to the source code of the commodity operating system, memory analysis tools can only resort to public symbols and documentations. As a result, these tools (e.g., Volatility [30]) can only identify documented objects (whose definitions are publicly available) and follow the pointers whose target types are also documented.

**(2) Ambiguous pointers.** Generic pointers (e.g., `void *`, `LIST_ENTRY`, and `struct list_head`) are prevalent in data structure definitions. It is hard to determine the exact target types for these generic pointers, and it is common for a generic pointer to have multiple type candidates. As a result, it is difficult to follow these generic pointers to identify the target objects. Pointers can also be dangling, and following the dangling pointers would lead to extraction of bogus objects.

**(3) Lack of robustness.** Because of such a low coverage, it is very easy for kernel attacks to evade memory analysis. Hiding an object can be as simple as manipulating the incoming links that are followed by the analysis tools. For example, FU rootkit [14] hides a process by unlinking the corresponding `EPROCESS` object from the active process list. To evaluate the validity of a memory object, the existing tools often rely on constraints that can be easily violated, such as pool tags, string constants, object lengths, etc. In Section 5.4, we demonstrate a synthetic attack that can completely defeat the utilities in Volatility by breaking these soft constraints.

Up to now, prior research efforts have been focused on tackling only one or two challenges above. No solution can address all the challenges in a holistic fashion. To improve robustness, several robust signature schemes have been proposed [11, 21]. These signature schemes can reliably detect important kernel objects by checking invariants (either strong value invariants or points-to relationship) in the kernel data structures. These signatures may not be distinct enough or may not even exist for many kernel objects (especially small ones). Therefore, we cannot rely on these robust signature schemes to achieve high coverage, not to mention that performance overhead is high for repeatedly searching signatures one by one throughout the memory.

Some efforts on data structure reverse engineering (such as REWARDS [22] and Howard [29]) may help extract kernel data structures definitions from commodity OSes. Potentially these system can help identify previously undocumented objects and links, and thus improve the coverage. However, these systems have only demonstrate their capabilities on relatively small user-level programs. Complete reverse engineering of kernel data structures is still a daunting task due to the complexity of the commodity OS kernel code and the kernel data structures.

In this paper, we present MACE[1], a holistic solution that meets all the following requirements:

(1) **Binary only approach.** MACE uses only the binary code of an OS, the public symbols, and documented data structure definitions to perform memory analysis. As a result, MACE is well suited for external forensic analysts to analyze closed-source OS like Windows.

(2) **Robustness.** To achieve high robustness, MACE relies on *only* points-to relations (or pointer constraints), which are generally hard to violate, to identify kernel objects. Furthermore, MACE evaluates both deterministic and probabilistic pointer constraints throughout the entire kernel memory space, to find a nearly optimal solution. Therefore, even if an attacker manages to manipulate some pointers, these "injected" errors would likely be corrected by the remaining pointers in the memory during this *global* evaluation process. Thus, the attack impact on the overall identification results is minimized.

(3) **High coverage and accuracy.** MACE can reconstruct a nearly complete kernel object graph, which consists of both documented and undocumented kernel object instances, and the connections among them. For undocumented objects, MACE can further discover certain type information for the pointer fields in these objects. For instance, MACE can identify function pointers and target types for data pointers.

(4) **Good efficiency.** MACE can scan a memory image and build a kernel object graph just a few minutes[2]. In contrast, the existing robust signature schemes [11, 21] use several minutes to only identify objects of a single type.

The core idea of MACE is to conduct supervised learning on pointers. That is, we first collect pointer constraints from a set of training memory images, in which kernel objects are correctly labeled by dynamic binary analysis. With the collected pointer constraints, we then perform probabilistic inference on pointers in an unlabeled memory image in a collective and correlative manner, to correctly label the pointers in the image. From these labeled pointers, we then reconstruct a nearly complete kernel object graph, for memory forensic purposes.

We leverage a key insight that the kernel object graph is a small-world network [13]: most kernel objects can be reached from other kernel objects within a few hops. A link from one object to another imposes a type constraint (either deterministic or probabilistic) on each side. The type constraint indicates the likelihood of a directly connected object to be of a particular type. The type constraints will accumulate and propagate to the objects that are not directly connected. Eventually, these constraints will be broadcast to the entire network until a convergence is reached.

We evaluated MACE for two closed-source operating systems: Windows XP SP3 and Windows 7 SP0 and found that MACE can achieve high recall and precision (95% and 96%, respectively) for Windows XP and Windows 7. The errors mostly come from undocumented objects and volatile memory allocations.

We further evaluated the performance of MACE on memory images infected with real-world malware samples to demonstrate how MACE facilitates kernel rootkit identification. With a more complete coverage of kernel objects, MACE recognized malicious function pointers in both documented and undocumented data structures, and detected hidden objects more reliably. At last, we devised two synthetic kernel attacks to show how fragile the existing memory analysis tools (such as Volatility) can be, and how resilient MACE is against these attacks.

## 2. PROBLEM STATEMENT & OVERVIEW

### 2.1 Problem Statement

Given a memory image, we aim to reliably identify nearly all the kernel objects and connections between them, without access to the OS source code. We rely on public symbols, public data structure definitions. This public knowledge is used by the existing memory analysis tools (e.g., Volatility [30]). We attempt to improve the coverage and robustness of these third-party memory analysis tools, by leveraging the same amount of knowledge.

In addition to identifying documented kernel objects, we also aim to extract partial knowledge of undocumented kernel objects. In particular, we would like to discover types of the pointer fields, including both data pointers and function pointers. This knowledge on pointers can help obtain a big picture of the entire kernel object graph and benefit security analysis on this graph (e.g., kernel rootkit detection). In other words, our goal is not to reverse engineer the kernel data structure definitions as REWARDS [22] and Howard [29], although our technique can be combined with these techniques to improve the quality of data structure reverse engineering.

We formalize the problem of kernel object labeling as follows: $M = \{m_i | 1 < i < |M|\}$ denotes the kernel memory space, where $m_i$ is the $i$th machine word and $|M|$ is the total number of machine words in the kernel address space. Our goal is to assign a label $l$ to each $m_i$. A label $l$ is defined as a pair of object type and offset $l = (t, o)$, where $t \in T$ and $o \in [0, \texttt{sizeof}(t))$. Here $T$ denotes the space of all object types.

To ensure high robustness, our solution cannot rely on soft constraints that can be easily manipulated by attackers, such as integer and string constants. For example, checking pool tags and the object size from the `OBJECT_HEADER` in Windows definitely helps verify the object types (both Volatility [30] and MAS [8] use this method to resolve type ambiguity). However, kernel rootkits can easily violate these soft constraints to evade and mislead these memory forensic tools. It means that our solution can only rely on pointer constraints, which are more difficult to tamper with. We should also anticipate that although complete sabotage of pointer constraints is not possible, attackers may manage to manipulate a certain amount of pointers. Therefore, our solution should tolerate pointer manipulation attacks to a certain degree.

[1]MACE stands for Memory Analysis through Correlative Evaluation.

[2]The current implementation of MACE is mostly in Python for quick prototyping. A C/C++ implementation would further reduce the analysis time to tens of seconds.

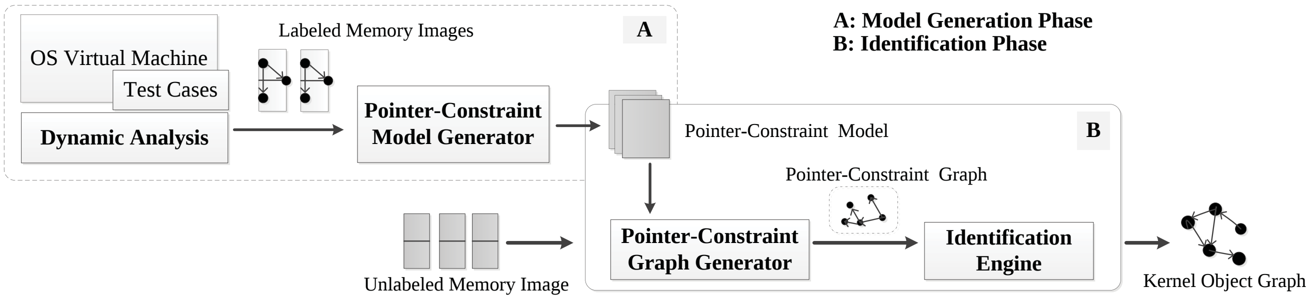


**Figure 1: System Overview. The model generation phase A outputs the pointer-constraint model. The identification phase B detects the kernel object graph on the unknown memory image.**

## 2.2 System Overview

We propose to take a probabilistic inference approach to label kernel objects based on their pointer constraints. Figure 1 depicts this workflow. Essentially, we propose a supervised learning technique. In the model generation phase, we perform dynamic binary analysis on the OS kernel to label kernel objects and learn a pointer-constraint model. Then in the identification phase, we will use this model to identify kernel objects in an unknown memory image.

**Model Generation Phase.** For a closed-source operating system (like Windows), we perform dynamic binary analysis to label kernel objects while the OS is running inside a virtual machine. These labeled kernel objects are then used to generate the pointer constraint model, which captures the probabilistic type constraints between pointer fields. To ensure the training is well-rounded, we conduct a set of test cases to exercise different components of the operating system, such as filesystem, network, IO, process/module/ thread management, etc. Consequently, the recorded memory images (with labeled kernel objects) capture diverse system states under these workloads.

If the source code of an OS is available, we could generate this pointer-constraint model in two ways. We could perform points-to analysis on the source code directly to generate extended type graph (as done in KOP [4] and MAS [8]), and then simply derive a model from the extended type graph. For a generic pointer, we would have to assign equal probability to each possible target type. To generate a model that better reflects the system states at runtime, we could also perform dynamic analysis described above. It means that if a target type for a generic pointer appears more often than the others at runtime, it would have higher probability. The model generated this way would lead to better classification results.

**Identification Phase.** Given the pointer-constraint model for one OS version and an arbitrary memory image of the same OS version, in the identification phase, MACE tries to identify kernel objects and their pointer relationships.

The problem of labeling pointers in the memory image based on the pointer constraints is equivalent to searching the optimal type assignment for each pointer under the given constraints. A plausible solution to this problem is Maximum Likelihood Estimation (MLE), wherein every possible assignment solution is enumerated and evaluated in terms of likelihood, i.e., the number of satisfied constraints. However, this solution proves to be NP-hard. Therefore, it is too expensive to iterate through all possible types for tremendous amount of pointers in the memory.

We approach this problem by using the *random surfer model* [16], which has been commonly used for complex networks, such as page ranking on the web [16]. Intuitively, in random surfer model, a score associated with each node in the graph is equivalent to the likelihood of this node being visited by the "random surfer". The likelihood of a node being visited is determined by how likely its neighbors are visited and how likely the "surfer" travels from a neighbor to this node. The random surfer model allows for effectively evaluating the scores for all nodes in the graph such that it is scalable even for very large graphs (e.g., the internet ). To the best of our knowledge, we are the first to apply the random surfer model to the problem of memory analysis.

In our problem domain, a node represents a labeled pointer, and an edge from one node to another dictates a confidence level that the source pointer has on the target pointer. In other words, the confidence level is a conditional probability on how likely the target pointer is correctly labeled given the source pointer is correctly labeled. We call this graph a *pointer-constraint graph*. We then apply the random surfer algorithm to calculate a nearly optimal score for each node (i.e., a pointer with a particular label). Finally, we compute object-level scores based on these pointers' scores and identify true kernel objects.

## 3. MODEL GENERATION

For a closed-source operating system, we generate the pointer-constraint model in two steps: 1) we conduct dynamic analysis to label kernel objects in the training data set; and 2) we learn the model by conducting statistical analysis on the training data.

## 3.1 Labeling Kernel Objects

We monitor the execution of the OS kernel and observe how kernel objects are allocated and de-allocated, and how these kernel objects are connected with each other. As we observe the actual binary execution of the OS kernel, we can obtain the ground truth, which is typically hard to get otherwise.

We leverage the dynamic analysis framework DECAF [17] to monitor the execution of an OS and construct the kernel data structure graph on the fly. In general, we monitor and label three kinds of kernel objects. We monitor kernel modules (e.g., `ntoskrnl.exe` and device drivers) by hooking `MmLoadSystemImage`. This is important because global data variables are located in these kernel modules. We hook `ObCreateObject` to monitor and label documented kernel objects (e.g., `EPROCESS`). Windows uses this function to create managed kernel objects (which are all documented). For other objects, we hook `ExAllocatePoolWithTag` and `ExFreePoolWithTag` to obtain a view of live memory objects in the dynamic memory pools. While there are other functions to allocate and free memory regions in the kernel, these two are the root functions. All the functions to be hooked are located in the main kernel component `ntoskrnl.exe`, and these functions' offsets can be obtained from the public symbol information.

In this way, we can precisely label kernel modules and documented kernel objects. However, for undocumented objects, we rely on their pool tags obtained from the `ExAllocatePoolWithTag` function call. This pool tag labeling mechanism is fairly common in the modern OSes. For example, SLAB in UNIX-like systems is a similar mechanism. Of course, several problems may arise if we simply label undocumented objects by their pool tags: 1) an object allocated with a pool tag may consist of multiple inner objects, which become invisible; and 2) objects of the same type may be allocated using several different pool tags. As reverse engineering undocumented objects is not only main goal, we accept these limitations and leave a better labeling approach as future work. For example, we could leverage the calling context of the memory allocation routine to label the object.

Certainly, these function hooks are specific to Windows. For another closed-source operating system, we will need to rely on its public documentation and public symbol information to find a set of functions to hook and label objects properly. The general principle should remain the same.

To recognize links between these kernel objects, we examine each double-word within each object and see if the value in the double-word falls in the memory region of any kernel object. If this is true, we treat this value as a pointer field and we establish a link between these two kernel objects. Note that in the kernel space, it is common for a pointer to point to the middle of a kernel object. This approach may lead to an overestimation in our study because a non-pointer field may happen to have a pointer-like value and thus be treated as a pointer. In practice, these pointer-like data fields will not affect the detection accuracy of MACE, because these noises are filtered out in the statistical analysis (described in Section 3.3). Moreover, pointer fields may not be 4-byte aligned in certain packed data structures, so we have to search double-words in all byte locations.

## 3.2 Test Cases

In order to ensure that the generated model has a diverse set of kernel objects, the test programs used for dynamic analysis need to activate different OS functionalities that are as diverse as possible. We include both standard OS benchmark and common software programs to be run in the guest OS to maximize the variety and number of kernel objects created. For the standard OS benchmark, we choose `lmbench` [1], as it performs several diverse actions in networking (TCP, UDP, RPC, and pipe), filesystem (file creation and deletion, cached file read, etc.), signal handling, memory access, etc. We also select several common and complex programs to further increase the training coverage, including web browsers, media players, word processors, and PDF readers.

## 3.3 Statistical Analysis

Without source code, we conduct statistical analysis to learn kernel objects and their relationships based on labeled memory images. Specifically, we utilize the pointer constraint model to represent the kernel objects and their relationships. The pointer constraint model includes offset constraints and target constraints.

**Offset Constraints:** An offset constraint dictates the pointer offset in the kernel object. For example, the offset constraint OC(A) represented in Figure 2(b) shows that Object A with 12 bytes has three pointers at offset 0, 4, and 8.

To learn offset constraints for each object type $t$, we go over all the instances of that object type and examine the pointer fields in them. An offset $o$ appears in the offset constraints of $t$ if and only if all the instances of $t$ have valid pointers at offset $o$. For example, we can learn that OC[A]=[0,4,8], since all instances of A in Figure 2(a) have pointer-like values at offset 0, 4 and 8.

**(a) Labeled Memory Image**

| Addr | Value | Label | Addr | Value | Label | Addr | Value | Label | Addr | Value | Label |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0x8000** | 0x80B0 | (A,0) | **0x8034** | 0x80A8 | (A,4) | **0x8068** | 0x80BC | (A,8) | **0x809C** | 0x1 | (A,c) |
| **0x8004** | 0x80A8 | (A,4) | **0x8038** | 0x80BC | (A,8) | **0x806C** | 0x1 | (A,c) | **0x80A0** | 0x1234 | (C,0) |
| **0x8008** | 0x80BC | (A,8) | **0x803C** | 0x1 | (A,c) | **0x8070** | 0x80B0 | (A,0) | **0x80A4** | 0x80B0 | (C,4) |
| **0x800C** | 0x1 | (A,c) | **0x8040** | 0x80B0 | (A,0) | **0x8074** | 0x80BC | (A,4) | **0x80A8** | 0xff | (D,0) |
| **0x8010** | 0x80B0 | (A,0) | **0x8044** | 0x80A8 | (A,4) | **0x8078** | 0x80BC | (A,8) | **0x80AC** | 0x80B0 | (D,4) |
| **0x8014** | 0x80A8 | (A,4) | **0x8048** | 0x80BC | (A,8) | **0x807C** | 0x1 | (A,c) | **0x80B0** | 0x1234 | (B,0) |
| **0x8018** | 0x80BC | (A,8) | **0x804C** | 0x1 | (A,c) | **0x8080** | 0x80B0 | (A,0) | **0x80B4** | 0x1 | (B,4) |
| **0x801C** | 0x1 | (A,c) | **0x8050** | 0x80B0 | (A,0) | **0x8084** | 0x80BC | (A,4) | **0x80B8** | 0x8000 | (B,8) |
| **0x8020** | 0x80B0 | (A,0) | **0x8054** | 0x80BC | (A,4) | **0x8088** | 0x80BC | (A,8) | **0x80BC** | 0xff | (E,0) |
| **0x8024** | 0x80A8 | (A,4) | **0x8058** | 0x80BC | (A,8) | **0x808C** | 0x1 | (A,c) | **0x80C0** | 0xff | (E,4) |
| **0x8028** | 0x80BC | (A,8) | **0x805C** | 0x1 | (A,c) | **0x8090** | 0x80A0 | (A,0) | **0x80C4** | 0x8008 | (E,8) |
| **0x802C** | 0x1 | (A,c) | **0x8060** | 0x80B0 | (A,0) | **0x8094** | 0x80BC | (A,4) | **0x80C8** | 0 | (D,0) |
| **0x8030** | 0x80B0 | (A,0) | **0x8064** | 0x80BC | (A,4) | **0x8098** | 0x80BC | (A,8) | **0x80CC** | 0x80B0 | (D,4) |

**(b) Generated Pointer Constraint Model**

| KOB | Offset Constraint | Target Constraint |
|---|---|---|
| | | TC[(A,0)] = {(B, 0, 0.9), (C, 0, 0.1)} |
| A: 0x10 | OC[A] = [0, 4, 8] | TC[(A,4)] = {(D, 0, 0.5),(E, 0, 0.5) } |
| B: 0x0c | OC[B] = [8] | TC[(A,8)] = {(E, 0, 1)} |
| C: 0x08 | OC[C] = [4] | TC[(B,8)] = {(A, 0, 1)} |
| D: 0x08 | OC[D] = [4] | TC[(C,4)] = {(B, 0, 1)} |
| E: 0x0c | OC[E] = [8] | TC[(E,8)] = {(A, 8, 1)} |
| | | TC[(D,4)] = {(B, 0, 1)} |

**Figure 2: An example of pointer-constraint model: (a) the labeled memory image for an OS version; (b) is the pointer-constraint model inferred from the labeled memory image. The first column of (b) means the object type and the size.**

**Target Constraints:** A target constraint is imposed on the target of a pointer field. It includes the target type and the probability indicating how likely the pointer target is of the particular type. The target constraint is also can be learned through statistical analysis. By iterating through all labeled pointer fields and their targets in the training memory dumps, we can compute these target constraints. For example, the statistical analysis on instances of object A in Fig. 2(a) learns that the pointer at offset 0 in object A has two target labels, $(B, 0)$ and $(C, 0)$. By counting the numbers of instances of A with different target types, we also compute the probabilities of $(B, 0)$ and $(C, 0)$ to be 0.1 and 0.9 respectively. The target constraint ($TC(A, 0)$) is shown in Fig. 2(b).

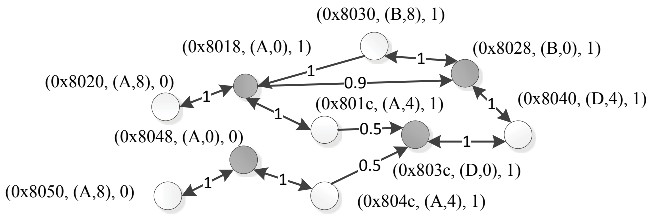
**Variable-Length Arrays.** The variable-length array is handled in the different manner, since its size is not a constant. We discover variable-length arrays using two conditions. (1) the size of a variable-length array is variable; (2) each entry of the array should have the same target type or a NULL pointer. This means that we only focus on object types with the variable size. For each object type with the variable size, we can determine the variable-length array by checking whether all entries of its instances share the same target type or zero. The arrays in our model will be labeled as "array" without the specific size. Its target offset constraint will be normalized to be relative to the start address of their hosting elements, instead of the base of the array.

## 4. KERNEL OBJECT IDENTIFICATION

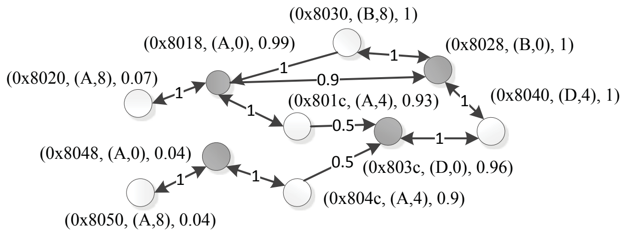
Given an arbitrary memory image, MACE tends to identify its kernel objects based learned constraint model from the training phase in the following steps: (1) it constructs a pointer-constraint graph from the memory image; (2) it applies the Random Surfer algorithm on the pointer-constraint graph until a convergence is reached; and (3) it selects true kernel objects based on the final scores on the pointer-constraint graph.

| Address | Value | Address | Value |
|---|---|---|---|
| 0x8018 | 0x8028 | 0x8038 | 0x4 |
| 0x801c | 0x803c | 0x803c | 0x7 |
| 0x8020 | 0x803c | 0x8040 | 0x8028 |
| 0x8024 | 0x1 | 0x8044 | 0x8 |
| 0x8028 | 0x11 | 0x8048 | 0x8034 |
| 0x802c | 0xc | 0x804c | 0x803c |
| 0x8030 | 0x8018 | 0x8050 | 0x8034 |
| 0x8034 | 0x6 | 0x8054 | 0x1 |

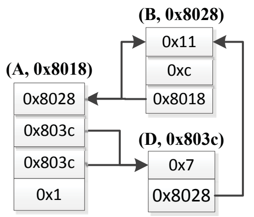
**(a) Unlabeled memory image**



**(b) Initial constraint graph**



**(c) Converged constraint graph**



**(d) Identified kernel object graph**

**Figure 3: An example for random pointer surfing. A solid node in the graph represents a pointer with offset 0, indicating the base of a kernel object.**

## 4.1 Pointer-Constraint Graph Construction

DEFINITION 1. *The pointer-constraint graph is a directed gra ph $G = (V, E)$. A node $v \in V$ is a tuple $(a, (t, o), r)$, where $a$ is the address of a pointer, $(t, o)$ labels the pointer as the object type $t$ and the offset $o$ with the object, and $r \in [0, 1]$ is the score indicating how likely this pointer is labeled correctly. An edge $e \in E$, $e = (u, v, r)$ represents the constraint from node $u$ to node $v$, where $r \in [0, 1]$ specifies the conditional probability how likely $v$ being correctly labeled if $u$ is correct.*

As an example, we show a pointer-constraint graph in Figure 3(b), which is constructed from an unlabeled memory image in Figure 3(a), using the model presented in Figure 2. Algorithm 1 describes how to construct a pointer-constraint graph.

We construct a pointer-constraint graph, starting with a number of "root" nodes, and then perform breadth-first traversal to add new nodes and edges into the graph. To find root nodes, we select kernel objects (e.g., EPROCESS and ETHREAD) that have many pointer fields inside and thus their offset constraints are fairly unique. We use these offset constraints to scan the kernel memory and find possible kernel objects. Then a root node is created for each pointer field of these objects. Some of these root nodes may be in fact wrong. Their scores may be updated during the evaluation of their target constraints. At last, we will rely on the random surfer algorithm (described in Section 4.2) to evaluate their authenticity. For example in Figure 3, we choose object A to find "root" nodes, since it has the most number of offset constraints. If we scan the memory image using A's offset constraint, we find two instances for A. Therefore, we create root nodes for object A. MACE will create a node for each offset constraint of object A. We assign 1 to node $(0x8018, (A, 0), 1)$, $(0x801c, (A, 4), 1)$, $(0x8020, (A, 8), 1)$, $(0x8048, (A, 0), 1)$, $(0x804c, (A, 4), 1)$ and $(0x8050, (A, 8), 1)$ because at this stage, their offset constraints are all met.

To further expand the constraint graph, we need a working queue $Q$ to perform this breadth-first traversal. To begin with, the root nodes are enqueued into $Q$. Then, on each iteration, a node $v$ is dequeued from $Q$. We retrieve from the model $PCM$ the target constraints $TC$ for this node $v$. If $v$ does have target constraints,

**Algorithm 1:** Pointer-Constraint Graph Construction

```
Input: Memory image $M$,Pointer-Constraint Model $PCM$
Output: Pointer-Constraint Graph $G$
$Q \leftarrow G.V$;
while $Q \neq \emptyset$ do
    $v \leftarrow Q.dequeue()$;
    $TC \leftarrow PCM.GetTC(v.t, v.o))$;
    if $TC \neq \emptyset$ then
        $Matched \leftarrow False$;
        for each $tc \in TC$ do
            if $PCM.CheckOC(M, M[v.a] - tc.o, tc.t) = True$
            then
                $Matched \leftarrow True$;
                $u \leftarrow (M[v.a], tc.t, tc.o, 1)$;
                if $u \notin G.V$ then
                    $G.AddNode(u)$;
                    $Q.enqueue(u)$;
                end
                $G.AddEdge(v, u, tc.r)$;
                $AddObjtoG(G, M[v.a] - tc.o, tc.t)$;
            end
        end
        if $Matched = False$ then
            $v.r \leftarrow 0$;
        end
    end
end
return $G$;
```

we go over each target constraint $tc$ in $TC$. $tc$ tells us a possible target label $(tc.t, tc.o)$ and its likelihood $tc.r$. Then, to check if this target label is compatible with the target memory, we check the offset constraints of the target type $tc.t$ and the memory region starting at the corresponding object base address $M[v.a] - tc.o$. This is done in $PCM.CheckOC()$ function. In other words, we check if the memory words at the offsets specified in the offset constraint are valid addresses. If this is not true, this target type can be not valid, so we check the next target constraint in $TC$. Otherwise, we extend $G$ into the target object.

To extend the graph $G$, we first check if the target node $u$ (with the same address and label) has been already created. If not, we validate that its pointer locations are compatible with its offset constraints. For all the nodes that pass the validation check, we create $u$ and set its initial score to 1. Otherwise we will set the score to be 0. We also enqueue $u$ to $Q$ for the subsequent breadth-first traversal. An edge $(v, u, tc.r)$ is added into $G$, where the edge's likelihood is obtained from the target constraint $tc$. For example, object A candidate fails the target checking at $0x8020$, $0x8048$ and $0x8050$, MACE has to update the value 1 to 0 for the node $(0x8020, (A, 8), 0)$, $(0x8048, (A, 0), 0)$ and $(0x8050, (A, 8), 0)$ in Figure 3(b).

Now we need to add the rest of pointers (if any) in the target object into $G$. To facilitate subsequent object-level classification, we always create a "base" node (whose offset is 0) for each object, even if the field at offset 0 is not a pointer. We further use this "base" node to bind all the pointers in that object by adding both incoming and outgoing edges between the "base" node. In this way, the scores on the pointers within one objects can flow back and forth to each other until a convergence is reached. In Figure 3, these "base" nodes are marked as solid circles.

If it turns out that none of the labels in target constraints of $v$ is compatible with the target memory region, we set its score $v.r$ to 0 because $v$'s label $(v.t, v.o)$ may be wrong. It is worth noting that we do not conclude that $v$ is absolutely wrong and remove it immediately from $G$. Note that a pointer in a true object may occasionally point to invalid target (or a new target that does not exist in our model). We leave it to the Random Surfer algorithm below to decide if this pointer is indeed labeled wrong.

## 4.2 Random Surfer Algorithm

We adopt the random surfer algorithm in [12] to find nearly optimal solution on the pointer-constraint graph. It is also proved to converge. Suppose $\mathbf{r}$ is a $|v|$-dimensional column vector called score vector, where $|v|$ is the number of nodes in $G$. $\mathbf{r}_i$ is the score of the $i$th node in $G$ (for the convenience we represent vectors and matrices in bold). Besides, we define a transition matrix $\mathbf{M}$, where $\mathbf{M}_{ij}$ is the transition probability from the $i^{\text{th}}$ node to the $j^{\text{th}}$ node in $G$. If there is no transition from $i$ to $j$, $\mathbf{M}_{ij}$ is assigned 0. Because the matrix $M$ is a stochastic matrix we normalize $\mathbf{M}$, such that each row of $\mathbf{M}$ sums to 1.

Equation 1 describes how to calculate the scores based on the neighbors' scores:

$$\mathbf{r}^{(k+1)} = (1 - \alpha - \beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{r}^{(0)} \tag{1}$$

where $\mathbf{p} = [\frac{\sum_i^{|v|} r_i}{|v|}]_{|v|\times 1}$ is a constant score vector, where $|v|$ is the number of node in $G$, and $\mathbf{r}^{(k)}$ indicates the score vector at iteration $k$. $\alpha$ is the damping factor used to jump out of isolated loops or clusters during surfing. In order to ensure that $\mathbf{r}^{(k)}$ finally converge, $\beta$ is introduced as another damping factor that controls the frequency jumping to the initial score distribution $\mathbf{r}^{(0)}$. Empirically, following [12] we set $\alpha = 0.7$, $\beta = 0.1$ to guarantee a good rate of convergence.

Algorithm 2 details how we update the score of each node in the constraint graph. For each iteration, the algorithm updates the score of each node based on scores of its neighbors and the constraints among them. After several iterations, the score of each node in $\mathbf{r}$ stabilizes. For each iteration, we calculate the mean square error between the current score vector and the previous one. If the error is smaller than the threshold $\epsilon$, we consider it to have converged. The final score vector will approximate a globally optimal solution that satisfies the constraint graph.

**Algorithm 2:** Random Surfer algorithm

```
input  : the transition matrix $\mathbf{M}$, the initial value vector $\mathbf{d}$, damping
         factor $\alpha$, $\beta$ and the vector constant $\mathbf{p}$
output: the converged score vector $\mathbf{r}$
$\mathbf{r}^{(0)} = \mathbf{d}$ ;
while $\delta \geq \epsilon$ do
    $\mathbf{r}^{(k+1)} = (1 - \alpha - \beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{d}$ ;
    $\delta = ||\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}||_2^2$ ;
end
```

Figure 3(c) shows the converged pointer-constraint graph after we applied Algorithm 2 on the graph in Figure 3(b). Although the object A at $0x8018$ fails the target checking at offset 8, its score is 0.99. The overall voting through the constraint graph still considers object A at $0x8018$ is more likely to be true. Although the offset constraints for object A at $0x8048$ show it could be A, the converged constraint graph tells us the score for A at $0x8048$ is 0.04. Therefore, $0x8048$ is impossible to be the base address for object A. This observation verifies that the label decision of an object type judges on overall situation rather the individual pointer constraint.

## 4.3 Kernel Object Labeling

The kernel object labeling utilizes k-means [31] method where $k = 2$ in our scenario to cluster base nodes in the constraint graph. $k = 2$ means that we only split labeled nodes of the same type into two sets including the true set and false set. The set with higher scores as true set means that all nodes in the set are correctly identified. In detail, the kernel object labeling clusters base nodes of same object type by their scores and generates the identified kernel object graph from the cluster with the higher score. For example, k-means splits (0x8018,(A,0),0.99) and (0x8048,(A,0),0.04) into two sets and we considers (0x8018,(A,0),0.99) as the true set. As for the example in Figure 3(d), we are able to classify the base nodes in the converged graph, and construct a kernel object graph. From the result, we can see that MACE can still find object A at $0x8018$, even if the constraint checking for the offset 8 at object A failed.

## 5. IMPLEMENTATION AND EVALUATION

The implementation of MACE comprises 3 components. One component performs an initial scan to recognize pointers on a memory image, which include one plugin (with 570 lines of Python code) to Volatility and one (with 78 lines of c code) to DECAF [17]. Another component is the plugin of DECAF with additional 800 lines of C code to gather the ground truth for kernel objects. The third component is a stand-alone Python program consisting of 6.2K LOC used for learning the pointer-constraint model and kernel object identification.

We evaluated MACE from the following aspects: Section 5.1 presents the model generation results, including how fast the model converges, how big the model is, and how long it takes to generate the model; Section 5.2 measures the accuracy and runtime performance of kernel object identification; Section 5.3 demonstrates MACE's capability of detecting rootkit footprints using realworld rootkit samples; Finally, Section 5.4 presents synthetic attacks demonstrating the attack tolerance of MACE over the other memory analysis tools.

**Experiment Setup.** We evaluate MACE on four sets of memory images: 1) 150 memory images from Windows XP Service Pack 3 including 100 images for training and 50 for detection (All these images are of 512 MB RAM); 2) 145 memory images from Windows 7 Service Pack 0 including 100 images for training and 45

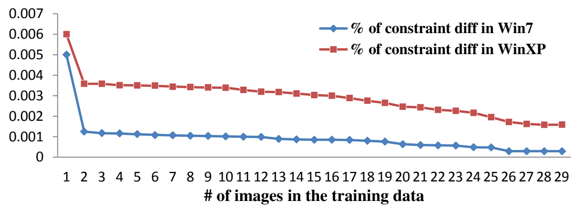


**Figure 4: Changes in the model quality $\delta$ (number of new target constraints + new offset constraints) across images. The small number of images can achieve a stable model.**

for detection (All these images are of 1.5 GB RAM); 3) 8 memory images of 512 MB RAM from kernel malware analysis; and 4) 1 memory image of 1.5 GB RAM for synthetic attack. The first 295 memory images are derived by using our dynamic analysis component (as discussed in Section 3.1). All experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2650 (2.00$GHz \times 8$) and 128 GB RAM running 64-bit Ubuntu 11.04.

## 5.1 Model Generation

**Model Convergence.** We generate a model from 100 memory images for Windows XP and Windows 7 respectively. In order to predict how close this model is to a theoretically perfect model, we randomly select $k$ images to generate another model, and compute a "diff" between these two models with respect to their offset constraints and target constraints. In this way, we can see how quickly the model converges when number of training images increases.

Figure 4 shows the model quality evaluation results for both Windows XP and Windows 7. It illustrates that as the number of images in the training set increases, the missing constraints of the model generated from $k$ images (compared to the model generated from 100 images) decreases exponentially and becomes stable very quickly. Even a model generated from one image is 99.4% similar to the model generated from 100 images.

**Model Generation Runtime.** Obtaining a labeled memory image took nearly 2 minutes for Windows XP and 3 minutes for Window 7. To speed up the image retrieving process, we run 10 virtual machines in parallel and each one conducts 10 different test cases. Finally, it takes 20 minutes on average to obtain 100 Windows XP images and 30 minutes on 100 Windows 7 images. For each image, the model generator extracts the pointer information from the memory image. It takes approximately 7 minutes per memory image for Windows XP and 15 minutes for Windows 7. This task can be finished within 40 and 80 minutes respectively for XP and Windows 7 by using 20 processors in parallel. In the end, it takes 20 minutes and 30 minutes to merge the results on 100 images and construct the pointer-constraint model for Windows XP and Windows 7 respectively. Overall, the model generation takes less than 2 and 3 hours for Windows XP and Windows 7 respectively.

## 5.2 Kernel Object Identification

We evaluate MACE's identification capabilities with respect to the accuracy and the runtime performance.

**Accuracy.** We use the two metrics, *Recall* and *Precision*, to measure the accuracy of the detection results. We calculate the recall and the precision using the following formulas:

$$Recall = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ in\ ground\ truth} \tag{2}$$

$$Precision = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ by\ MACE} \tag{3}$$

| Steps | Time (Sec) | |
|---|---|---|
| | Windows XP | Windows 7 |
| **Initial Scan** | $3.0 \pm 2.1$ | $7.0 \pm 4.9$ |
| **Graph Generation** | $180 \pm 2.0$ | $315 \pm 5.6$ |
| **Kernel Object Inference** | $22 \pm 0.7$ | $55 \pm 1.6$ |
| **TOTAL** | $205 \pm 4.5$ | $377 \pm 12.6$ |

**Table 1:** MACE**'s Identification Runtime Performance**

We measured the above two metrics over 45 memory images for Windows XP and 7 respectively. Figure 5 shows MACE can achieve good identification results for both Windows XP SP3 and Windows 7 SP0. More specifically, MACE achieves the 95% recall on average and 98% precision on average for Windows XP SP3, and the 96% recall and 95% precision on average for Windows 7 SP0. The detection result of MACE is close to KOP [4] which relies on the source code. Furthermore, we observed zero false negatives and false positives in the kernel objects of high forensic values (the ones extracted by Volatility). 5% false negatives are from the undocumented objects, such as the objects with pool tags 'IoNm' and 'GH0<'. 2% false positives are caused by the undocumented kernel objects of small sizes, such as 'Mmpv'.

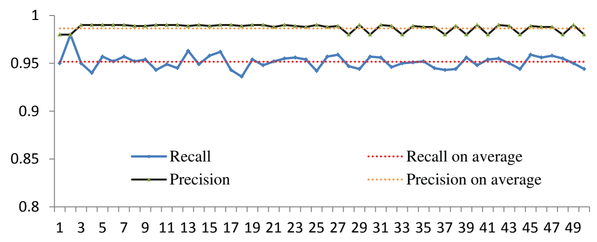
**Runtime Performance.** We evaluated MACE's identification runtime performance in two scenarios. In the cloud computing scenario, the virtual machine state has been loaded in memory, so scanning through the virtual machine memory is very fast. We used virtual machine snapshots (50 Windows XP SP3 memory images with 512 MB RAM and 45 Windows 7 SP0 images with 1.5 GB RAM) from KVM/QEMU to evaluate this scenario. On the contrary, in the memory forensics scenario, the memory content is first dumped into a file, then the forensic analysis is performed on the file. The analysis for this scenario will be slower, due to the time for loading the file into memory and other factors. The result for the first scenario is shown in Table 5.2.

As we see, MACE finished the kernel object identification in 205 seconds for Windows XP and 377 seconds for Windows 7 on average. The identification for Windows 7 takes longer, as the memory images for Windows 7 are larger and contain more kernel objects. Note that our current implementation is in Python mainly for fast prototyping. The identification performance can be significantly reduced to tens of seconds for a C/C++ implementation.

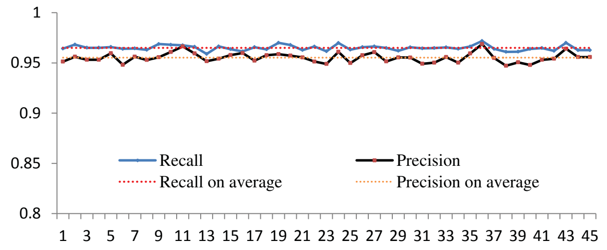
## 5.3 Detecting Kernel Rootkit Footprints

As a case study, we show how to use the kernel object graph constructed by MACE to detect kernel rootkit footprints. To this end, we developed a tool to analyze memory images infected with kernel rootkits. Using the kernel object graph constructed by MACE, the tool can detect malicious function pointers and hidden objects in the infected images. For the sake of fair comparison, this tool follows the similar logic as SFPD and GHOST that were built on top of KOP [4]. Note that KOP requires the source code of Microsoft Windows to construct the extended type graph and traverse the kernel objects, whereas MACE does not. From the viewpoint of external memory analysts, we would like to see whether the tool built on top of MACE can reach the same detection performance as the ones built on KOP.

More specifically, to detect malicious function pointers, our tool iterates through all the kernel objects and examine the function pointers in them. MACE can differentiate function pointers from data pointers, because the target of a function pointer must be located in the text section of a kernel module, which can be determined from parsing the headers of that module. As the pointer-constraint model contains all the valid targets for each function



(a) Windows XP SP3



(b) Windows 7 SP0

**Figure 5: Precision and Recall.**

pointer during the training phase, to determine a malicious function pointer, we can check whether the actual target of this function pointer does not belong to any of the valid targets. To be consistent to SFPD, our tool also excludes manipulations in System Service Dispatch Tables (SSDTs) and Interrupt Descriptor Table (IDT).

To detect hidden objects, we use Volatility as the reference system. In particular, we use common commands in Volatility, such as `pslist`. Then we compare these objects obtained from Volatility and the kernel object graph from MACE. If a kernel object of one of the above types appears only in the result from MACE, it is deemed a hidden object.

We collected 8 memory images infected with various real world kernel rootkits. Two images (Stuxnet.vmem and ds_fuzz_hidden _proc.img) were downloaded from the Volatility Google Code website. The rest of memory images (including TDSS, Fakeuinit, ZeroAccess, Papras, Haxdoor, and FuTo) were recorded by running these samples separately in a virtual machine. It demonstrated that MACE can tolerate small changes in the kernel code and the locations of global pointers, making it practical to analyze realworld memory images. We list rootkit detection results in Table 5.2.

**Malicious Function Pointers.** It is not surprising that our tool built on MACE can detect malicious function pointers in the common data structures like `_DRIVER_OBJECT`, etc. The other rootkit detection tools would have the same coverage, assuming that they can identify these data structures correctly. More interesting results are found for TDSS and Fakeuinit, and they are highlighted in the table. For TDSS, we found two malicious function pointers located in the data section of "ntoskrnl.exe". With help of IDA Pro [18], we confirmed that tampering with these two function pointers can effectively hook `IofCompleteRequest` and `IofCallDriver` and thus manipulate the communication between the main kernel and the device drivers. As for Fakeuinit, we found 5 malicious function pointers in undocumented kernel objects, whose pool tags are 'NDpp', 'NDpb and 'NDmo' respectively. Through manual investigation, we determined that these kernel objects are operated by `NDIS.sys` (the central networking module in Windows) to manage the network stack. Manipulating these function pointers can effectively intercept the network communication. Here, note that these pool tags identified from our model are not from the infected memory images. So even if the actual pool tags are modified by the rootkits, our detection results would stay unaffected.

**Hidden Objects.** Our tool also detected hidden processes and modules for several kernel rootkits. It shows that instead of just examining several known linked lists and tables, MACE discovers kernel objects in a global scope. We notice that a recently developed tool (`psxview`) in Volatility can detect hidden processes, by checking other data structures in addition to the active process linked list. In comparison, our tool checks `_EPROCESS` objects in the entire kernel data structure graph, not only the ones publicly known to the memory analysts, not to mention that our tool can also detect other kinds of hidden objects.

## 5.4 Attack Tolerance

To evaluate the attack tolerance of MACE, We devised two synthetic attacks: 1) pool tag manipulation; and 2) deterministic pointer removal.

**Pool Tag Manipulation.** This synthetic attack is as simple as modifying pool tags for the objects like `_KDBG`, `_EPROCESS`, etc. After these modifications, the Windows system continued to run properly, indicating that there is no integrity check on pool tags in Windows. Then we tested the commands in volatility, and none of them output any results. The commands like `psscan` and `thrdscan` rely on pool tag as a constraint to scan particular kinds of kernel objects, so simple modifications on pool tags can easily sabotage these commands. Other commands like `pslist` and `threads` also failed, even though they did not use pool tag as a constraint extensively to scan kernel objects. The failure of these commands is due to the missing `_KDBG`. These commands must scan `_KDBG` to determine the right Windows version and thus locate the right start address of the relevant data structures. For the same reason, the new `psxview` command also failed. In contrast, MACE was not affected by this synthetic attack at all, because by design MACE does not use pool tags and other kinds of soft constraints to identify kernel objects.

**Deterministic Pointer Removal.** We suppose that an attacker can manage to remove a fraction of pointers to hide certain kernel objects, without causing a system crash. In particular, we would like to see how MACE's identification performance degrades while a fraction of deterministic pointers are removed, because all the existing tools only examine deterministic pointers. Furthermore, we simulate a "strawman" system as the theoretical upper-bound for any memory analysis system that only examines deterministic pointers. This "strawman" system starts with global variables in the data sections of kernel modules, and only follows deterministic pointers and always makes a right decision whether an object is valid even when some of its pointers or pointer targets are invalid.

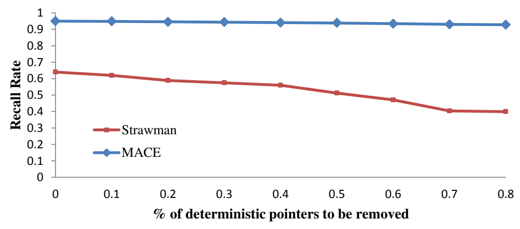


**Figure 6: Recall Degradation on Link Sabotage Attacks**

| Name | Malicious Location | # | Cat. |
|---|---|---|---|
| Backdoor: W32TDSS | **ntoskrnl.exe:0x7c484** | 1 | M |
| | **ntoskrnl.exe:0x7c480** | 1 | M |
| | _GENERIC_CALLBACK.Callback | 2 | M |
| | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| | _LDR_DATA_TABLE_ENTRY | 1 | H |
| stuxnet.vmem | _GENERIC_CALLBACK.Callback | 1 | M |
| | _NOTIFICATION_PACKET. NotificationRoutine | 1 | M |
| | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _DRIVER_OBJECT.MajorFunction[] | 3 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| Trojan-Spy.Win32. Fakeuinit.a | **NDpp:0x18** | 1 | M |
| | **NDmo:0x38** | 1 | M |
| | **NDmo:0x50** | 1 | M |
| | **NDmo:0x40** | 1 | M |
| | **NDpb:0x4c** | 1 | M |
| | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _DRIVER_OBJECT.DriverUnload | 1 | M |
| | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _ETHREAD.StartAddress | 1 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| Backdoor. Win32. ZAccess.dl | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _GENERIC_CALLBACK.Callback | 1 | M |
| | _ETHREAD.StartAddress | 1 | M |
| TrojanPSW. Win32.Papras | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _DRIVER_OBJECT.DriverUnload | 1 | M |
| | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| ds_fuzz_ hid den_proc.img | _EPROCESS | 7 | H |
| | _DRIVER_OBJECT.DriverUnload | 1 | M |
| | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _DRIVER_OBJECT.MajorFunction[] | 4 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| Win32. Haxdoor | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _DRIVER_OBJECT.MajorFunction[] | 2 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| RootKit: Futo | _DRIVER_OBJECT.DriverStart | 1 | M |
| | _DRIVER_OBJECT.DriverInit | 1 | M |
| | _DRIVER_OBJECT.DriverUnload | 1 | M |
| | _DRIVER_OBJECT.MajorFunction[] | 4 | M |
| | _LDR_DATA_TABLE_ENTRY.EntryPoint | 1 | M |
| | _EPROCESS | 1 | H |

**Table 2: Rootkit Footprints Detected By** MACE**. In the column of "Category", M means "malicious function pointer", and H stands for "hidden object".**

To evaluate this attack, we randomly remove a fraction of deterministic links (e.g, 10%, 20%, 30%, etc.) from a labeled memory image, and compute the recall for both MACE and the strawman system. For the strawman system, we consider it would miss a kernel object if there is no deterministic path from a kernel module to it. Figure 6 presents this result for a Windows XP image. It shows how the recall degrades more or less with the increase of the percentage of sabotaged deterministic pointers.

We can see that even when the attack is absent (0% pointers are removed), the recall for the strawman system is only 65%, demonstrating the necessity of incorporating non-deterministic pointers into the analysis. The performance of the strawman system degrades to about 40% when 80% deterministic pointers are removed. This result appears to be reasonable. However, this is just a theoretical upper-bound. The real memory analysis systems that only follow deterministic pointers will certainly perform worse than it. Further, the strawman system failed to identify many important kernel objects. For example, out of 22 process objects, it missed 20.

In contrast, the recall degradation for MACE is barely noticeable even when 80% deterministic pointers are removed, thanks to the small-world effect of the kernel object graph and the global evaluation nature of random surfer model.

## 6. DISCUSSION

In this section, we discuss several potential and practical issues and concerns related to MACE. Also, we make clarifications and suggest countermeasures if necessary.

**Kernel Patches.** By design, we need to train one model for each OS version. Kernel patches introduce changes in the main kernel module and possibly certain data structure definitions. So we would need to train a new model for every single kernel patch. In reality, it is not necessary, because the changes introduced in these patches are usually small. The major kernel data structure definitions remain unchanged. As demonstrated in our experiment, we can still use the model generated for Windows XP Service Pack 3 to analyze two memory images downloaded from the Volatility website and obtain good results. In these two memory images, patches have been applied to Windows XP Service Pack 3.

**Third-party Device Drivers.** A memory image under the analysis may have third-party device drivers loaded, which have not been observed during model generation phase. In this case, MACE will not be able to identify kernel objects defined in these device drivers. However, MACE will still detect these device drivers, because a number of documented objects (e.g., `DEVICE_OBJECT`) will be created for them and they will be detected by MACE. It is still an open research problem to discover data structures in these third-party device drivers. We leave it as future work.

## 7. RELATED WORK

**Memory Analysis Frameworks.** Memory analysis came into limelight after 2004 work by Carrier et al. [5]. There exist plenty of open-source and commodity memory analysis tools [2, 6, 9, 10, 19, 23–25, 27, 28, 30, 32].The memory analysis is also extended to the analysis of hypervisors and virtual machine [15].

**Robust Signature Schemes.** To improve robustness for memory analysis [26], two signature schemes [11, 21] have been proposed. These two signature schemes detect kernel objects by relying on invariants that are hard to be manipulated and evaded. Although the work by Dolan-Gavitt et al. [11] is based on data invariants, most of the identified invariants are indeed on pointer fields. In comparison, MACE leverages the insights in pointer invariants and takes a fresh look into the problem of memory analysis.

**Source Code based Memory Analysis.** The knowledge of data structure definitions can be directly obtained from the kernel source code. SigGraph [21] extracts points-to relationships directly from the Linux source code and creates pointer-based signatures. SigGraph only extracts deterministic points-to relationships, and omits generic pointers. In order to obtain a nearly complete data structure graph, KOP [4] and MAS [8] perform points-to analysis on the Windows kernel source code. They identify points-to relationships for generic pointers and generate extended type graph. In contrast, MACE is designed for external forensic analysts who often do not have access to the kernel source code of an investigated system.

**Probabilistic Memory Analysis.** Several systems also take probabilistic approaches in memory analysis. Laika [7] applies Bayesian unsupervised machine learning algorithm to infer a type graph from a memory snapshot of a user-level program execution. In comparison, the inference algorithm used in MACE is supervised learning. In the training set, the kernel objects are classified and labeled using dynamic analysis. Some assumptions made in Laika do not hold in kernel data structures. For example, Laika assumes a pointer should point to the beginning of an object. This is not true for kernel data structures in common OSes like Windows and Linux.

To identify data structure instances that have been freed, DIMSUM [20] takes a probabilistic inference approach. Given a data structure definition, DIMSUM constructs a factor graph and computes marginal probabilities of all the candidate memory locations that satisfy the data structure constraints. In comparison, MACE tackles a similar problem but in a larger scale. The computational overhead would be too high to compute marginal probabilities for all pointers.

## 8. CONCLUSION

In this paper, we presented MACE, a memory kernel object mining tool that can accurately identify kernel objects in a robust manner. We evaluated MACE on 100 memory images for Windows XP SP3 and Windows 7 SP0. The experimental results showed that MACE can achieve the recall of 95% and the precision of 98% on average. Furthermore, the experiment also demonstrates the the robustness and the good efficiency. To illustrate the strength of MACE, we also conducted synthetic attacks on a memory image from Window XP SP3. The detection result showed that MACE outperformed other external memory analysis tools with respect to wider coverage and better robustness.

## 9. ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their comments, and insightful discussions with Dr.Weidong Cui. This research was supported by NSF grants #1018217 and #1054605, AFO SR grant #FA9550-14-1-0119, McAfee Inc, and VMware Inc. Any opinion, findings, conclusions, or recommendation are those of the authors and not necessarily of the funding agencies.

## References

[1] LMBench – Tools for Performance Analysis. http://www.bitmover.com/lmbench.
[2] N. Beebe. Digital forensic research: The good, the bad and the unaddressed. In *Advances in Digital Forensics V*. 2009.
[3] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, 2012.
[4] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*, 2009.
[5] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.
[6] A. Case, L. Marziale, and G. G. Richard, III. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.
[7] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.
[8] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of USENIX Security Symposium*, 2012.
[9] B. Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digital Investigation*, 4:62–64, 2007.
[10] B. Dolan-Gavitt. Forensic analysis of the windows registry in memory. *Digital Investigation*, 5:S26–S32, 2008.
[11] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security(CCS'09)*, 2009.
[12] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM Journal on Scientific Computing*, 27(4):1181–1201, 2006.
[13] Q. Feng, A. Prakash, H. Yin, and Z. Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. Technical Report SYR-EECS-2014-05, Syracuse University, 2014.
[14] FU Rootkit. http://www.rootkit.com~/project.php?id=12, 2005.
[15] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor memory forensics. In *Proceedings of Symposium on Research in Attacks, Intrusion, and Defenses (RAID'13)*, 2013.
[16] T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.
[17] A. Henderson, A. Prakash, L. K. Yan, et al. make it work, make it right, make it fast. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.
[18] The IDA Pro Disassembler and Debugger. http://www.datarescue.com/idabase/.
[19] J. D. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4(1):24–29, 2007.
[20] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th ISOC Network and Distributed System Security Symposium (NDSS'12)*, 2012.
[21] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.
[22] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.
[23] MANDIANT Memoryze. http://www.mandiant.com/resources/download/memoryze.
[24] S. Mrdovic, A. Huseinovic, and E. Zajko. Combining static and live digital forensic analysis in virtual environment. In *Proceedings of XXII International Symposium on Information, Communication and Automation Technologies, 2009.*
[25] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006.
[26] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.
[27] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3:10–16, 2006.
[28] A. Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64, 2008.
[29] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.
[30] Volatility: Memory Forencis System. https://www.volatilesystems.com/default/volatility/.
[31] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained k-means clustering with background knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML'01)*, 2001.
[32] R. Zhang, L. Wang, and S. Zhang. Windows memory analysis based on kpcr. In *Proceedings of the Fifth International Conference on Information Assurance and Security( IAS'09)*, 2009.