Tracking Rootkit Footprints with a Practical Memory Analysis System

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Zhilei Xu
Massachusetts Institute of Technology
timxu@mit.edu

Ellick Chan
University of Illinois at Urbana-Champaign
emchan@illinois.edu

Abstract

In this paper, we present MAS, a practical memory analysis system for identifying a kernel rootkit's memory footprint in an infected system. We also present two large-scale studies of applying MAS to 848 real-world Windows kernel crash dumps and 154,768 potential malware samples.

Error propagation and invalid pointers are two key challenges that stop previous pointer-based memory traversal solutions from effectively and efficiently analyzing real-world systems. MAS uses a new memory traversal algorithm to support error correction and stop error propagation. Our enhanced static analysis allows the MAS memory traversal to avoid error-prone operations and provides it with a reliable partial type assignment.

Our experiments show that MAS was able to analyze all memory snapshots quickly with typical running times between 30 and 160 seconds per snapshot and with near perfect accuracy. Our kernel malware study observes that the malware samples we tested hooked 191 different function pointers in 31 different data structures. With MAS, we were able to determine quickly that 95 out of the 848 crash dumps contained kernel rootkits.

1 Introduction

Kernel rootkits represent a significant threat to computer security because, once a rootkit compromises the OS kernel, it owns the entire software stack which allows it to evade detections and launch many kinds of attacks. For instance, the Alureon rootkit [1] was infamous for stealing passwords and credit card data, running botnets, and causing a large number of Windows systems to crash. Kernel rootkits also present a serious challenge for malware analysis because, to hide its existence, a rootkit attempts to manipulate the kernel code and data of an infected system.

An important task in detecting and analyzing kernel rootkits is to identify all the changes a rootkit makes to an infected OS kernel for hijacking code execution or hiding its activities. We call these changes a rootkit's *memory footprint*. We perform this task in two common scenarios: We detect if real-world computer systems are infected by kernel rootkits. We also analyze suspicious software in a controlled environment. One can use either execution tracing or memory analysis in a controlled environment, but is usually limited to memory analysis for real-world systems. In this paper we focus on the memory analysis approach since it can be applied in both scenarios.

After many years of research on kernel rootkits, we still lack a *practical* memory analysis system that is *accurate*, *robust*, and *performant*. In other words, we expect such a practical system to correctly and quickly identify all memory changes made by a rootkit to arbitrary systems that may have a variety of kernel modules loaded. Furthermore, we lack a large-scale study of kernel rootkit behaviors, partly because there is no practical system that can analyze memory infected by kernel rootkits in an accurate, robust and performant manner.

In this paper, we present MAS, a practical memory analysis system for identifying a rootkit's memory footprint. We also present the results of two large-scale experiments in which we use MAS to analyze 837 kernel crash dumps of real-world systems running Windows 7, and 154,768 potential malware samples from the repository of a major commercial anti-malware vendor. These are the two major contributions of this paper.

Previous work [2, 3, 19] has established that, to identify a rootkit's memory footprint, we need to check not only the integrity of kernel code and static data but also the integrity of dynamic data, and the real challenge lies in the latter task.

In order to locate dynamic data, these systems first locate static data objects in each loaded module, then recursively follow the pointers in these objects and in all newly identified data objects, until no new data object can be added. Unlike the earlier systems, KOP [3] includes *generic pointers* (e.g., *void*∗) in its memory traversal, and shows that failing to do so will prevent the memory traversal from reaching about two thirds of the dynamic objects.

Previous solutions do not sufficiently address an important practical problem of this memory traversal procedure: *its tendency to accumulate and propagate errors*. A typical large real-world kernel memory image is bound to contain *invalid pointers*. That is, there are likely to be dynamic objects with pointer fields not pointing to valid objects. Following such pointers results in objects being incorrectly included in the object mapping. Worse, such identification errors can be propagated due to the nature of the recursive, greedy memory traversal. A single incorrectly identified data object may cause many more mistakes in the subsequent traversal.

Invalid pointers may exist for a variety of reasons. For example, an object may have been allocated, but not yet initialized. KOP is exposed to a second source of potential errors. KOP tries to follow all generic pointers. If the pointer type cannot be uniquely determined, KOP tries to decide the correct type using a heuristic. A fraction of these guesses are bound to be incorrect.

In light of these problems, we design MAS to control the number of errors that arise from following invalid pointers and to contain their effects. Instead of performing a greedy memory traversal that is vulnerable to error propagation, MAS uses a new traversal scheme to support error correction. MAS also uses static analysis to derive information that can be used to uniquely identify many objects and their types without having to rely on the recursive traversal procedure. Furthermore, MAS is not subject to errors caused by *ambiguous pointers*, i.e., pointers whose type cannot be uniquely determined. It uses an enhanced static analysis to identify unique types for a large fraction of generic pointers and ignores all remaining ambiguous pointers. While this may reduce coverage, it will never cause an object to be recognized incorrectly. Our evaluation will show that the impact on coverage is minor. Finally, before accepting an object, MAS checks a number of constraints, including new constraints we derive from our static analysis.

We implemented a prototype of MAS and compared it with KOP on eleven crash dumps of real-world systems running Windows Vista SP1. MAS's performance is one order of magnitude better than KOP regarding both static analysis and memory traversal. MAS did not miss or misidentify any function pointers found by KOP, but KOP missed or misidentified up to 40% of suspicious function pointers (i.e., function pointers that point to untrusted code).

In our large-scale experiments, we ran MAS over crash dumps taken from 837 real-world systems running Windows 7 and memory snapshots taken from Windows XP SP3 VMs subjected to one of 154,768 potential real-world malware samples. For the Windows 7 crash dumps, MAS took 105 seconds to analyze a single dump on average. It identified a total of about 400,000 suspicious function pointers. We were able to verify the correctness of all but 24 of them. Moreover, with the results of MAS, we were able to quickly identify 90 Windows 7 crash dumps (and five Windows Vista SP1 crash dumps) that were infected by kernel rootkits. In our study of malware samples, MAS required about 30 seconds to analyze each VM memory snapshot. Our study shows that the kernel rootkits we tested hooked 191 function pointer fields in 31 data structures. It also shows that many malware samples had identical footprints, which suggests that we can use MAS to detect new malware samples/families that have different memory footprints.

The rest of this paper is organized as follows. Section 2 provides an overview of the paper. Sections 3 and 4 describe the design of MAS and explain the algorithms used for static analysis and memory traversal. Section 5 explains how we evaluate the set of objects found by MAS for suspicious activity. Section 6 describes our implementation of MAS. Section 7 describes our evaluation of MAS. Section 8 and Section 9 describe two large-scale experiments in which we analyze malware samples and identify rootkits from crash dumps. Sections 10 and Section 11 discuss related work and limitations. Finally, Section 12 concludes the paper.

2 Overview

The goal of MAS is to identify all memory changes a rootkit makes for hijacking execution and hiding its activities. MAS does so in three steps: static analysis, memory traversal, and integrity checking.

Static Analysis: MAS takes the source code of the OS kernel and drivers as the input and uses a pointer analysis algorithm to identify candidate types for generic pointers such as *void*∗ and linked list constructs. Furthermore, it also computes the associations between data types and pool tags [18].

Memory Traversal: MAS tries to identify dynamic data objects in a given memory snapshot. Besides the snapshot, the input includes the type related information derived from static analysis and the symbol information [15] for each loaded module (if it is available).

Integrity Checking: MAS identifies the memory changes a rootkit makes by inspecting the integrity of code, static data and dynamic data (recognized from memory traversal). In addition to checking if some code section is modified, MAS detects two kinds of violations: (1) a function pointer points to a memory region outside of a list of known good modules; (2) a data object is hidden from a system program. The list of identified integrity violations is the final output of MAS. Such information can be used to detect if a system is infected by a rootkit or analyze a rootkit's behavior.

Next we describe these three steps in detail.

3 Static Analysis

In this section, we present our demand-driven pointer analysis algorithm. After that, we describe how we use this algorithm to identify candidate types for generic pointers and data types associated with pool tags.

3.1 Demand-Driven Pointer Analysis

We use *demand-driven* pointer analysis because we do not need the alias information for all the variables in a program, which traditional pointer analyses compute. Instead, we only compute the alias sets of generic pointers, a small portion of all the variables in a program.

Our demand-driven pointer analysis follows largely the approach of Zheng and Rugina [27]. Since our goal is to *precisely* identify candidate types for generic pointers, we extend Zheng and Rugina's pointer analysis to be field-sensitive, context-sensitive and partially flow-sensitive. We achieve partial flow-sensitivity by converting a program to the Static Single Assignment (SSA) form conservatively. We enforce context-sensitivity in a way similar to [23]. We handle indirect calls in our analysis as well.

Next we will summarize the approach of [27] and provide a detailed description of our extension to field sensitivity.

3.1.1 Program Expression Graph

The algorithm of [27] operates on a *Program Expression Graph* (PEG), a graph representation of all expressions and assignments in a C-like program. In this paper, we represent an expression as a C variable with $*$ (for the dereference operation), & (for the take-address operation) and $\rightarrow$ (for the field operation). In a PEG, the nodes are program expressions, and the edges are of two kinds:

Assignment Edge (A): For each assignment $e_1 = e_2$, there is an A-edge from e_2 to e_1.

Dereference Edge (D): For each dereference $*e$, there is a D-edge from e to $*e$; for each address &e, there is a D-edge from &e to e.

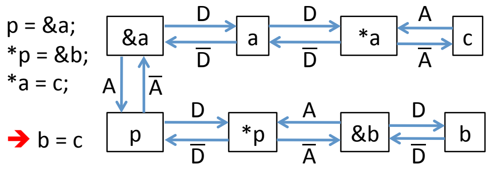

Figure 1: Sample program and its PEG

For each A and D edge, there is also a corresponding *inverse* edge in the opposite direction, denoted by $\overline{A}$ and $\overline{D}$. The edges can also be treated as relations between the corresponding nodes; so relations $\overline{A}$ and $\overline{D}$ are the *inverse relations* of A and D. Figure 1 shows a sample program and its PEG.

3.1.2 CFL-Reachability

In addition to the A and D relations (edges), we further define two relations between expressions (nodes):

Value Alias (V): If a and b may evaluate to the same value, we say they are value aliases, represented as aVb.

Memory Alias (M): If the addresses of a and b may denote to the same location, we say they are memory aliases, represented as aMb.

Given an interesting expression p, our pointer analysis searches for the set of expressions q such that pVq. We call this the*value alias set* of p. Similar to [27], we formulate the computation of the V relation as a Context-Free Language (CFL) reachability problem [21] over the program expression graph. Specifically, a relation R over the nodes of a PEG can be formulated as a CFL-reachability problem by constructing a grammar G such that a node pair (a,b) has the relation R if and only if there is a path from a to b such that the sequence of labels along the path belongs to the language $L(G)$. The context-free grammar G_V for value and memory alias relations is:

$$\begin{array}{lccl} \text{Value Aliases:} & V & ::= & M \mid M\overline{A}V \mid VAM \\ \text{Memory Aliases:} & M & ::= & \varepsilon \mid \overline{D}VD \end{array}$$

The grammar G_V has *non-terminals* V and M, and *terminals* $A, \overline{A}, D$, and $\overline{D}$. Readers can verify that the sample PEG in Figure 1 contains a path from b to c with label sequence $\overline{D}A\overline{D}\overline{A}DD\overline{A}$ that can be produced by the V non-terminal in G_V. So the grammar successfully deducts that b and c are value aliases. The intuition behind each production rule is:

$M ::= \varepsilon$ a is a memory alias of itself.

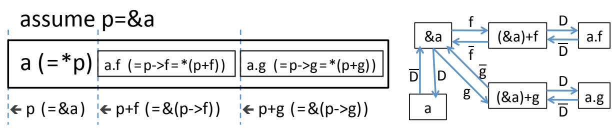

Figure 2: The relation of struct, field, and base pointer; and the corresponding PEG representation.

$M ::= \overline{D}VD$ Given $*p\overline{D}pVqD*q$ then, because p and q are value aliases, $*p$ and $*q$ are memory aliases.

$V ::= M$ Memory aliases are also value aliases.

$V ::= VAM$ Given $aVbAcMd$, the value of a propagates to c, which may reside in the same memory as d. Thus, a and d are value aliases. Similarly $V ::= M\overline{A}V$.

Given this grammar, Zheng and Rugina go on to construct a hierarchical state machine and design an algorithm that decides whether two expressions are memory aliases. They also sketch an extension of the alias analysis algorithm for computing the value alias set of a single expression, which we adopt in MAS. Next, we describe how we extend the basic grammar to achieve field sensitivity.

3.1.3 Field Sensitivity

Field-sensitivity is necessary for our pointer analysis since we want to distinguish a generic pointer field from other fields in the same data structure. Field-insensitive analysis, on the other hand, treats all fields in a data structure as the structure itself.

Fields in C can be represented by means of *pointer arithmetic*: given a base pointer p and a field f, $\&(p \rightarrow f)$ is the *field pointer* which points to a field inside the structure $*p$. We use $p+f$ to denote $\&(p \rightarrow f)$, and $p+f$ is in fact the result of offsetting p by a fixed number of bytes determined by field f. The other constructs $p \rightarrow f$ $(= *(p+f))$ and $a.f$ $(= (\&a) \rightarrow f = *(\&a+f))$ are merely syntactic sugar, as shown in Figure 2.

To support field-sensitivity in our pointer analysis, we first add edges to the PEG to represent the field relations. For every field descriptor, we create a field label f_i. Then for each base pointer p, if its field pointer $p+f_i$ exists in the program, we add an edge labeled f_i from p to $p+f_i$ and an inverse edge $\overline{f_i}$ in the opposite direction. As shown in Figure 2.

Zheng and Rugina [27] suggest adding $V ::= \overline{f_i}Vf_i$ to the grammar G_V for field-sensitivity. With this addition, the grammar becomes:

$$\begin{array}{lll} M & ::= & (\overline{D}VD)? \\ V & ::= & M \mid (M?\overline{A})^*V(AM?)^* \mid \overline{f_i}Vf_i \end{array}$$

```
typedef struct {
  void *header; // call this field F
  int   status;
} KOBJECT;
KOBJECT *x, *y;
*x = *y;
```

Figure 3: Example code of struct assignment.

However, we observe that this is insufficient to track all the value aliases because of a feature in C called *struct assignment*. One can assign a structure to another as if they were both simple variables, and the effect is the same as doing assignments between corresponding fields recursively (because each field can possibly be an embedded structure).

Figure 3 shows a simple example where handling struct assignment becomes crucial to the analysis. $x \rightarrow header$ and $y \rightarrow header$ are value aliases, as well as $x \rightarrow status$ and $y \rightarrow status$. However, the extended grammar suggested by Zheng and Rugina can not capture these alias relationships correctly. The relevant edges connecting from $x \rightarrow header$ to $y \rightarrow header$ produce the label sequence $\overline{Df}D\overline{AD}fD$, which cannot be generated from the "V" non-terminal in Zheng and Rugina's extended grammar. Struct assignment is a common feature widely used in various programs. We must handle it properly when computing value aliases.

Struct assignment can only happen when the two variables involved are of the same type, and that type is precisely known to the compiler. Taking advantage of this property, we have an effective and efficient fix for Zheng and Rugina's algorithm. In the program expression graph, we expand each struct assignment to the individual assignments of all corresponding fields. In the example code, $*x = *y$ is expanded to $x \rightarrow header = y \rightarrow header; x \rightarrow status = y \rightarrow status$. If some field is an embedded struct, then this expansion is done recursively, eventually down to the "leaf" fields. The program expression graph built this way is free of struct assignment, and Zheng and Rugina's extended grammar works properly on this kind of PEG.

3.2 Type Candidate Inference

We have implemented Zheng and Rugina's algorithm with our extension to do demand-driven pointer analysis. MAS uses this pointer analysis to derive the set of type-related information for identifying dynamic data object in memory traversal. The set of type-related information has two parts: candidate types for generic pointers and candidate types for pool tags [18] Note that we use candidate types and type candidates interchangeably. Next we will describe how we derive them in detail.

3.2.1 Candidate Types of Generic Pointers

A generic pointer is a pointer whose type definition does not reveal the actual type of the data it refers to. In MAS, we consider two kinds of generic pointers: *void∗* and pointers in linked list constructs. We consider linked list constructs because the declared type of its pointer fields does not reflect the larger data structure it links together in the list.

For an expression p of type *void∗*, its candidate types are the set of types of its value aliases. For instance, given $FOO * q$; $void * p$; $p = q$, we get p's candidate type as $FOO*$. To derive the candidate types for a pointer field f_i of type *void∗*, we need to consider all its instances. Thus, f_i's candidate types are the set of types of all the value aliases of the pointer field's instances in the form of $X \rightarrow f_i$.

We need to solve two problems to compute candidate types for pointer fields in linked list constructs. First, we are concerned with the larger data structures that are linked together in a list. When a linked list pointer field's value alias is in the form of $\&(a- > f_i)$, we say its *nested* candidate type is $\&(A- > f_i)$ where a's type is $A*$. This nested candidate type allows us to identify the larger data structure A when the linked list pointer points to its field f_i. For simplicity, we still use candidate types when we discuss linked list constructs.

Second, the head node and the entry nodes in a linked list tend to have different data structures. If we do not differentiate them, the candidate types of a linked list pointer field will have both types, which causes unnecessary type ambiguity. To solve this problem, we leverage the semantics of APIs for linked list constructs. For instance, *InsertTailList* is a function in Windows [16] for inserting an entry at the tail of a doubly linked list. It takes two parameters, *ListHead* and *Entry*. To differentiate the list head and entry, we compute the value alias sets of *ListHead/InsertTailList* and *Entry/InsertTailList*, where $a/func$ represents the parameter a of a function $func$. Then we match value aliases from each set based on the call stack. For each valid pair of $\&(a \rightarrow f_i)$ and $\&(b \rightarrow f_j)$, we derive that a list head at $\&(A \rightarrow f_i)$ has a nested candidate type of $\&(B \rightarrow f_j)$ where a's type is $A*$ and b's type is $B*$. This approach requires prior knowledge of all linked list constructs and their APIs. Given the limited number of such constructs, it is not a hurdle for adapting MAS to large programs like the Windows kernel and drivers.

To control the number of candidate types, we apply three refinement techniques to the basic algorithm. First, for every linked list pointer p, MAS excludes all value aliases q of p if q's type is different from p. This is because we did not observe any link list pointers being converted to other types, and such value aliases are almost always false positives introduced by imprecise analysis. Second, for each pointer path from p to its value alias q, we check if it involves a type cast to *void∗*. If so, we will ignore the path. We do this for two reasons: the type before the cast has already revealed the candidate type, and we avoid the noisy aliases following the type cast. Third, when there are multiple candidate types, we look for the largest common nested types among all candidate types. If such a common nested type exists, we use it as the single candidate type. In the example shown in Figure 4, the largest common nested type of struct A and struct B is struct C.

```
struct A {
   struct C ac;
   struct D ad;
};
struct B {
   struct C bc;
   struct E be;
};
```

Figure 4: Example of a common nested type.

3.2.2 Candidate Types of Pool Tags

In recent Windows operating systems, pool tags [18] are used to track memory allocations of one or more particular data types by a kernel component. A pool tag is a four-character literal passed to the pool manager at a memory allocation or deallocation. One such API is *ExAllocatePoolWithTag*. For many pool tags, a memory block with a particular pool tag is always allocated for a unique data type. For instance, "Irp " is always for the data type *IRP*. In MAS, we use static analysis to automatically unearth the associations between a pool tag and data types and use them in our memory traversal. We call the types associated with a pool tag the candidate types for the pool tag. Note that such associations are not limited to Windows. In the Linux kernel, the slab allocator is used to provide specialized per-type allocations. In this paper, our design and implementation are focused on supporting Windows kernel pool management. But the techniques can be easily ported to support Linux kernel memory management.

Our approach for computing pool tag's type information is similar to the approach used for linked list constructs. Taking *ExAllocatePoolWithTag* as an example, we first compute the value alias sets for *return/ExAllocatePoolWithTag* and *Tag/ExAllocatePoolWithTag*, where the former represents the return value of *ExAllocatePoolWithTag* and the latter is the pool tag parameter. Since pool tags are usually specified directly at function calls for memory allocations, we do a simple traversal by following assignments on the program expression graph to compute the "value alias" set of $Tag/ExAllocatePoolWithTag$. Then we match the value aliases in each set based on the call stack. For instance, given the following code, our analysis will infer that the pool tag 'DooF' is associated with the type FOO.

$FOO *f = (FOO*)\ ExAllocatePoolWithTag(\ NonPagedPool, sizeof(\ FOO\ ), 'DooF');$

4 Memory Traversal

In this section, we describe how MAS locates dynamic data objects in a given memory snapshot and identifies their types. The inputs to this step include the memory snapshot, the type related information derived from static analysis, and the symbol information [15] for each loaded module in the memory snapshot (if it is available).

The basic memory traversal in MAS is similar to previous work [2, 3, 19]. It first locates the static objects in each loaded module based on the symbol information, then performs a breadth-first traversal by following pointers in the static objects and all newly identified data objects until no new object is added. MAS follows generic pointers for which our static analysis was able to derive a unique type. In the absence of a robust method for resolving multiple type candidates during memory traversal, MAS ignores all ambiguous pointers.

In order to increase coverage, MAS uses the associations between a pool tag and data types that may appear in memory blocks labeled with this tag. We directly identify data objects (i.e., without following a pointer) when a pool tag is only associated with a single data type.

Invalid pointers are common in kernel memory for many reasons. There may be a lag between the time a pool block is allocated and the time it is initialized. Also, a dangling pointer may point to a pool block that was freed and allocated again for different use. There exist even data objects that are partially initialized due to performance optimizations (or programming errors).

Our solution to invalid pointers have two main components: constraint checking and error correction. We only add a new data object during memory traversal or during type assignment based on pool tags if it satisfies the following constraints.

- Size Constraint: a data object must lie completely within a memory block. (We collect the information of all allocated memory blocks before the memory traversal.)

- Pointer Constraint: a data object's pointer fields must either be null or point to the kernel address range.

- Enum Constraint: a data object's enum fields must take a valid enum value which is stored in the PDB files.

- Pool Tag Constraint: the type of a data object must be in the set of data types associated with the pool block in which the data object is located.

KOP [3] only checks size and pointer constraints, which is not effective for smaller sized objects since they tend to have fewer pointer fields and fit into most memory blocks. The checking of pool tag constraints allows MAS to mitigate this problem.

A final constraint states that two incompatible objects cannot occupy overlapping addresses. We say two overlapped objects are type compatible if their overlapped parts have equivalent types (i.e., with the same memory layout after being expanded into primitive types and pointers). For example, one object may be a sub structure of the other object. We check this constraint before accepting an object. A violation of this constraint is a clear indication that an error has been made or is about to be made. Either the new object or the existing object that collides with it must be wrong.

We select one of the two objects based on several confidence criteria. Objects that we found without following pointers, such as global variables or objects identified through pool tags, are not subject to invalid pointer errors. We always select such objects over other objects. If both objects were found by following pointers, we select the larger object, since we typically check more constraints for larger objects. If the decision is to reject the existing object, we also remove all objects that were added by following its pointers recursively and cannot be reached from other objects.

5 Integrity Checking

The last step in identifying a kernel rootkit's memory footprint is to perform integrity checking. The inputs to integrity checking include the memory snapshot, the list of data objects identified from memory traversal, the pdb and image file of each loaded module when it is available. Note that the set of image files serves as the white list of trusted code.

A rootkit tampers with kernel memory for two main purposes: run its own code and hide its own activity. To do so, a rootkit either hijacks kernel execution by modifying code or function pointers or directly manipulates kernel data. MAS checks three kinds of integrity as follows.

- Code Integrity: trusted code in memory should match with the image file on disk.

- Function Pointer Integrity: function pointers should point to the trusted code.
- Visibility Integrity: data objects found by MAS should be visible to system tools (e.g., those available in a debugger for listing processes and modules).

The visibility integrity checking allows MAS to report *hidden objects* such as hidden processes and hidden modules. For instance, to find hidden processes, MAS uses a debugger command (e.g., !*process*) to get the list of processes in a memory snapshot, then compares it with the process objects found by memory traversal. If a process object is not in the list returned by the debugger command, it is marked as a hidden process. To check function pointer integrity, MAS inspects not only well known hooking points such as the system call table but also each function pointer in the data objects identified from the memory traversal. Function pointers that point to a memory region outside of the trusted code are reported as *suspicious function pointers*. Violations of code integrity are reported as *suspicious code hooks*.

MAS can be used in two scenarios: detect if a real-world system is infected by rootkits or analyze the behavior of a malware sample in a controlled environment. If the white list of trusted code is complete, any integrity violation can be automatically attributed to rootkit infection. It is trivial to construct such a complete list based on a copy of a clean system in a controlled environment. However, when checking real-world systems, such a complete list may be available in some cases (e.g., machines inside an enterprise or virtual machines in a cloud) but not always. When the list of trusted code is incomplete, we will need an expert to inspect integrity violations reported by MAS before deciding if a system is infected. We will report our experiences of detecting rootkits from real-world crash dumps in Section 9.

6 Implementation

We implemented MAS with 12,000 lines of C++ code for the static analysis and 24,000 lines of code for memory traversal and integrity checking.

For static analysis, we developed a PREfast [14] plugin to extract information from the AST trees generated by the Microsoft C/C++ compiler. We implemented the pointer analysis as a stand alone DLL that, upon request, computes the value alias set for a given program expression based on the information extracted by the PREfast plugin. Since our pointer analysis is demand-driven and can run in parallel, we implemented our type candidate lookup to take advantage of that. We run a separate parallel job for each generic pointer. After all parallel jobs are done, we merge the inferred type relations together. We implemented the parallel type candidate lookup on a cluster running Windows HPC Server 2008 R2 [17].

For analyzing memory snapshots, the key logic was implemented as an extension of WinDbg [13]. In addition, we implemented a DLL based on the Debug Interface Access SDK [12] to programmatically access the symbol information stored in PDB files [15].

During memory traversal, we frequently access two kinds of data, allocated memory blocks and data objects identified, where a memory block may contain multiple data objects and no two data objects overlap in memory. We use a multi-level data structure in MAS in order to obtain fast *store* and *retrieve* operations for the two kinds of type data. At the bottom level, we use a page-table like data structure to achieve fast lookup for an arbitrary address. Here a hash table simply based on the starting addresses of allocated memory blocks cannot meet our need because a given memory address may fall into the middle of a memory block. Given a memory address, if there exists a memory block that covers it, the lookup in the bottom-level structure returns a pointer to a data structure that stores all the information for the memory block. In this data structure, we use a sorted list to store all the data objects identified in the memory block. We choose a sorted list because the number of data objects on a single memory block is small.

To speed up type check, we maintain a cache of matched subtypes and their offsets for each aggregate type and check the cache first before doing the type consistency check in a brute force way. We choose to use a cache because, for an aggregate type, type consistency checks usually occur repeatedly for a small number of its nested types.

7 Evaluation

This section evaluates the accuracy, robustness and performance of MAS. We perform the evaluation on three sets of memory snapshots: (a) 154,768 memory snapshots derived from our large scale kernel malware analysis; (b) a set of 837 real-world crash dumps from end user machines running Windows 7; (c) a set of 11 real-world crash dumps from end user machines running Windows Vista SP1. The last set of Windows Vista SP1 crash dumps allowed us to compare MAS directly to KOP [3]. For our analysis on real-world crash dumps, the white list of trusted code contains all the binaries available on Microsoft's symbol server. For our analysis of malware samples, the white list of trusted code contains all the binaries from a clean VM image. Our experiments were conducted on a machine running Intel Xeon Quad-Core 2.93 GHz with 12 GB RAM unless specified otherwise.

Id	Size (MB)	Modules	Fct. ptrs. MAS	Fct. ptrs. KOP	FP. KOP	FN. KOP
1	245	154	64	43	22	21
2	149	144	55	47	28	8
3	305	203	673	N/A	N/A	N/A
4	270	157	257	236	37	21
5	247	159	75	45	19	30
6	127	125	46	38	9	8
7	315	157	283	265	30	18
8	250	141	105	97	26	8
9	204	144	50	40	26	10
10	255	141	167	157	24	10
11	312	203	235	189	11	46

Table 1: Results on eleven Windows Vista SP1 crash dumps. "Fct. ptrs." represents the number of function pointers correctly identified by MAS or KOP.

7.1 Accuracy and Robustness

The goal of this section is to evaluate the accuracy and robustness of MAS. We face the general difficulty that it is hard and time consuming to obtain an object mapping that is known to be correct (i.e., ground truth) even in a controlled environment. For the real-world crash dumps for which we had no data beyond the crash dumps themselves, it appears unclear if and how a ground truth could be established. Given these methodological difficulties, much of the evidence we present in this section has to be indirect.

Our first data set consists of the outputs of MAS on the 837 Windows 7 crash dumps. We tried to establish whether the function pointers reported by MAS as suspicious are indeed function pointers. We inspected whether the target of the function pointers appeared to be the beginning of a function. The vast majority of function pointer targets contained a small set of code patterns corresponding to function preambles. This allowed us to automate most of pointer checks by running a program that checks for these patterns. We inspected the remaining pointers manually. We applied a second criterion to the function pointers whose targets did not appear to be code. We accepted all function pointer candidates that were fields in objects whose existence could be derived directly and unambiguously from the symbol information. This included global variables and objects that could be reached from global variables by following only uniquely determined typed pointers. This left us with a total of 24 dubious pointers out of total of $398,987$ function pointers that MAS had output.

The eleven Windows Vista SP1 crash dumps in our data set allowed us to perform a direct comparison with KOP. We examined manually all discrepancies between the outputs of MAS and KOP. KOP appeared to suffer from both false positives and false negatives (see Table 1). We first examined all function pointers returned by MAS and found that they are valid. Then we examined manually the targets of all function pointers reported by KOP that had not been output by MAS. None of the targets appeared to be the start of a function. Thus, we classified these pointers as false positives for KOP (FP. KOP in Table 1). We also observed a number of function pointers that were found by MAS, but not by KOP. Since we had concluded that the targets of these pointers are function entry points, we classified them as false negatives for KOP (FN. KOP in Table 1). KOP missed as much as 40% of the function pointers found by MAS. Furthermore, KOP as much as 40% of the function pointers reported by KOP appear to be incorrect.

We also tried to interpret the function pointers returned by MAS. A large fraction of the reported function pointers appeared to point to third-party drivers that were not included in our static analysis. However, in addition to detecting the footprints of widely used anti-virus software, we also found clear signs of rootkit infections in five out of the eleven crash dumps. We will discuss how we detect rootkits in real-world crash dumps in Section 9.

Next, we attempted to estimate the internal consistency of the objects found by MAS. We examined the complete kernel object mappings produced by MAS for inconsistent pointers. These are pointers whose type is incompatible with the object type that the object mapping has assigned to the pointer's target. For example, an object mapping might contain an object of type T_1 at address A. Another object in the mapping might contain a pointer P of some other type $T_2 \neq T_1$ that also points to A. P is an inconsistent pointer. Such inconsistencies may exist even if the object mapping is error free because of invalid pointers in objects and because of memory corruptions in the crash dump. But they may also indicate errors in the object mapping, for example as a result of following invalid pointers. We call an object inconsis-

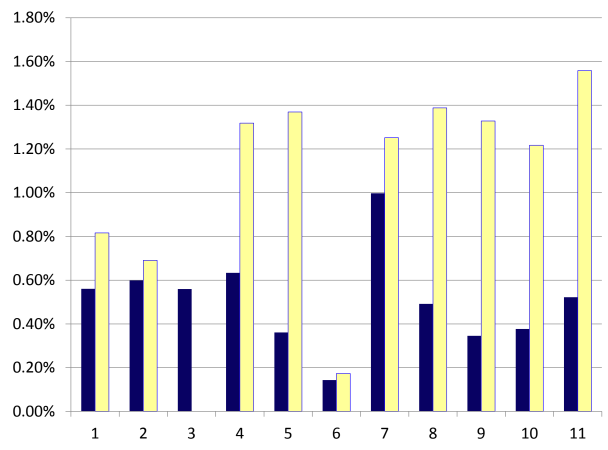

Figure 5: Percentage of inconsistent objects in the object mappings for MAS (left) and KOP (right). KOP did not produce a result for the third dump.

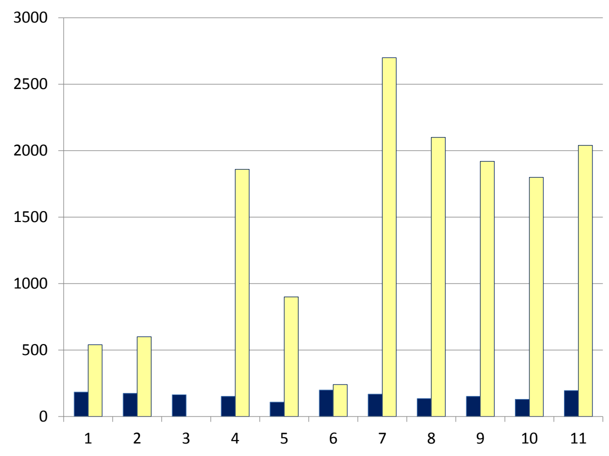

Figure 6: Running times in seconds of MAS (left) and KOP (right) on eleven real-world Windows Vista SP1 crash dumps. KOP did not produce a result for the third dump.

tent if it is the target of at least one inconsistent pointer. Figure 5 displays the percentage of inconsistent objects in the object mappings found by MAS and KOP for the Windows Vista SP1 crash dumps. We consider this number to be an indication of the correctness of the object mapping. On average, the object mappings produced by MAS contain 0.5% inconsistent objects. This number is 1% for the objects mappings produced by KOP.

7.2 Performance

This section evaluates the running time of MAS.

Static Analysis We performed the static analysis for Windows XP SP3, Windows Vista SP1 and Windows 7. Our evaluation is focused on Windows Vista SP1 since it allows us to compare MAS and KOP directly. The static analysis on Windows Vista SP1 includes the Windows kernel and a set of 63 standard drivers (such as win32k, ntfs and tcpip). This is the same set of drivers analyzed by KOP. The code base has 3.5 million lines of code. The program expression graph has 2.2 million nodes and 7.3 million edges. MAS performed almost 23,000 candidate type lookups.

We performed the static analysis on a 100 node cluster running Windows Server 2008 R2 HPC Edition, where each node has two Quad-Core 2.5 GHz Xeon processors with 16 GB RAM. Each node was used to perform 228 candidate type lookups. The whole static analysis took less than 5 hours. The corresponding time for KOP reported in [3] is 48 hours on a somewhat older, single processor machine.

The key advantage of MAS over KOP is that MAS's static analysis can run in parallel. This allows MAS to finish the static analysis in 5 hours on 100 nodes. On the other hand, the combined machine time of 500 hours is much larger than KOP's running time. This is partly because MAS does not achieve perfect parallelization. For instance, it takes 0.5 hour to load the program expression graph into memory on every node; alias analyses for indirect calls are computed on demand on each node and thus are not shared, which causes repeated computations as well. Furthermore, MAS converts a program to the Static Single Assignment (SSA) form conservatively, which increases the computation.

Dynamic Analysis Next, we report on the total running times of memory traversal and integrity checking of MAS on three sets of memory snapshots. Figure 6 displays the running times of MAS and KOP on the eleven Windows Vista SP1 crash dumps. On average, MAS (160 seconds per dump) is more than 9 times faster than KOP (24.5 *minutes* per dump). KOP failed to terminate on crash dump 3 within the two hour time limit we had set.

Figure 7 displays the distribution of MAS's running times on the 837 Windows 7 crash dumps. The running times are concentrated between 40 and 160 seconds. The average running time is 105 seconds, and 99.9% of all runs complete in less than 5 minutes.

Finally, the average running time of MAS on the 154,768 memory snapshots from our large-scale malware study is 31 seconds. The running time distribution is highly concentrated around this value.

In summary, our experiments demonstrate that MAS can quickly and accurately analyze real-world crash dumps as well as memory snapshots of virtual machines.

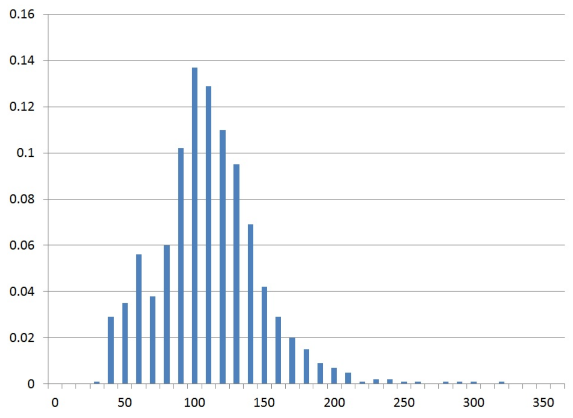

Figure 7: Running time (in seconds) distribution of MAS on 837 real-world Windows 7 crash dumps.

When compared directly, MAS was nearly an order of magnitude faster than KOP. MAS did not misidentify or miss any functions pointers found by KOP in the eleven Windows Vista SP1 dumps, but KOP missed or misidentified as much as 40% of the suspicious function pointers.

8 Kernel Malware Study

In this section we present the results of our study of a large collection of 154,768 potential malware samples that we obtained from a major vendor of anti-virus software. These samples originated from a variety of sources. Their behavior was unknown to us. This included the question whether a sample even contained malware. All samples were different types of Windows binaries: executables (.exe), dynamically linked libraries (.dll) and drivers (.sys).

We used MAS to analyze the samples. More precisely, for each sample, we booted a clean Windows XP SP3 VM with 256 MB of RAM and one virtual processor and loaded and executed it. We ran .exe's directly. We ran .dll's with the help of a standard executable that loads a dll and causes its DllMain function to be executed. We loaded drivers (.sys) using the service control manager (sc.exe). After launching the sample, we waited for one minute, then took a memory snapshot of the VM, converted it into a Windows crash dump and ran MAS over the crash dump.

In order to gain additional insight into the events that take place in the VM, we wrote a driver that makes most of the kernel address space of the VM not executable (by setting the corresponding bits in the page tables) and catches and records any non-execute (NX) page faults. The driver also records the loading and unloading of kernel modules and the allocation and deallocation of pool blocks. We loaded the driver in the VM before launching the sample.

We used a 25 node compute cluster to evaluate all 154,768 samples. The cluster nodes were running Windows Server 2008 R2. We used Hyper-V as our Virtual Machine Monitor. On each cluster node, we ran between 4 and 8 VMs, running a total of 164 VMs simultaneously at any time. Each job ran for 2 to 3 minutes. Since the VM jobs were I/O bound we took a number of measures to manage disk traffic: The VMs used differencing disks based on a single base image. We interleaved the startup of VMs such that the I/O intensive phases at the beginning and end of some jobs coincided with the one minute idle period of other jobs. All 154,768 jobs completed in less than 48 hours.

MAS reported kernel behaviors for only 89,474 of the samples. We analyzed the events recorded by our driver for the remaining 65,294 samples for which MAS had output no results. The driver logs showed that, in all but 1286 cases, neither module loading nor non-executable page faults were recorded. For the 1286 samples, the driver logs showed that no non-executable page faults were detected, and some modules were loaded after the sample was launched but all of the modules had been unloaded before the memory snapshot was taken. Based on this evidence, it appears that the memory snapshots for which MAS reported no results did not contain any data that MAS should have reported.

There are several potential reasons for the relatively large number of samples without reportable kernel behaviors. As stated above, some of the samples may simply not have been malware. Also, the crude way in which we launch the samples may have caused samples to fail to execute. It may also have caused malware not to become active. Techniques for reliably triggering malware have been studied elsewhere [5, 8] and are not the focus of this paper. The rest of this section presents the results of our analysis for the 89,474 samples for which MAS reported kernel behaviors.

8.1 General Behavior Statistics

Table 2 displays counts on the different categories of kernel behavior we observed. The count for a category is the number of samples that displayed behavior in that category. Some samples displayed behaviors in more than one category. Most categories correspond to modifications of static data structures that can be detected with existing tools. *IDT* represents modifications to the function pointers in the processor's interrupt descriptor table. *Sysenter* represents modifications to the hardware register that determines the target address of a `sysenter` instruction. *Callgate* represents similar modifications to function pointers in hardware-defined call gate structures.

Category	Count
IDT	20
Sysenter	1
Callgate	23
Syscall Table (SSDT)	3652
Hidden Process	1476
Hidden Module	43828
Code Hooks	17744
Module Imports and Exports	103
Function Pointer	84051

Table 2: Distribution of malware behaviors.

The next group of categories represents static software-defined function pointers. The system call table (SSDT) is a table of function pointers to the individual system call handler functions. *Hidden process* and *hidden module* stand for attempts to hide processes or modules by removing them from the data structures Windows maintains to keep track of processes and loaded modules. *Code Hooks* represent modifications of legitimate executable code. *Module Imports and Exports* represent tampering with the function pointers in the import and export lists of loaded modules.

Finally, the *Function Pointer* category includes modifications to function pointers in data objects found in MAS's memory traversal. Most of the objects are dynamic data (i.e., reside in the kernel pool) and some of them are from global variables. This is by far the most frequent category. About 94% of the samples display this behavior in some form. Since this is also the one category for which existing tools provide at best limited information, we examined it in more detail.

8.2 Function Pointer Hooking

We found that the samples were hooking a total of 191 unique function pointer fields from 31 different data structures belonging to the Windows kernel and five drivers (ntfs, fastfat, ndis, fltmgr, null). Figure 8 shows the number of samples that hooked each of the 191 function pointer fields. We observe a high concentration on a small set of pointers and a long tail. The two plateaus between 0 and 60 correspond mostly to function pointers from `nt!_DRIVER_OBJECT` and `nt!_FAST_IO_DISPATCH`. Almost 50% of the function pointers were hooked by only one or two samples.

We also counted the number of distinct dynamic function pointers hooked by each sample. The distribution is displayed in Figure 9. It is highly concentrated. Almost half the samples hook exactly 32 function pointers. There is a smaller concentration around the value 4. This high concentration suggests that versions or exact copies of the same underlying malware are present in a large number of samples. We further investigated this observation by clustering the samples.

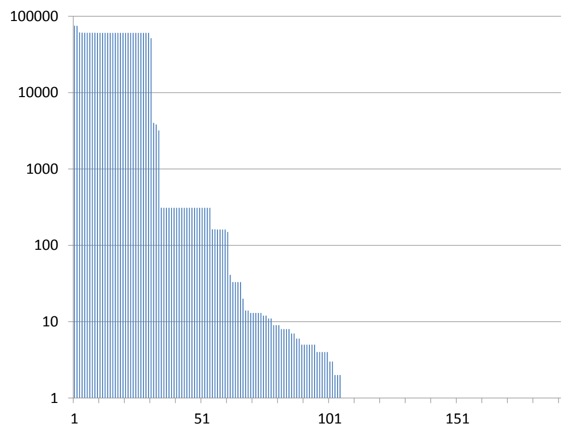

Figure 8: Number of samples that hooked each of the 191 different function pointers for which MAS detected hooking.

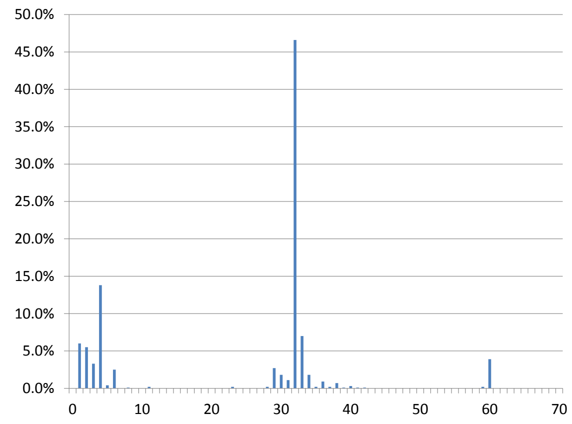

Figure 9: Distribution of the number of dynamic function pointers hooked by each sample

8.3 Clustering

To cluster samples, we first extracted the following information from MAS's report as a sample's footprint. For each suspicious function pointer, we use a tuple including "FUNCPTR" (indicating this tuple is about function pointers), function pointer field name, and data structure name. To differentiate the cases when different known drivers are hooked, we replaced the data structure name ("nt!_DRIVER_OBJECT") with a driver name (e.g., "\Driver \disk") for known drivers. For each code hook, we use a tuple including "CODEHOOK", module name, function name, offset, and the number of

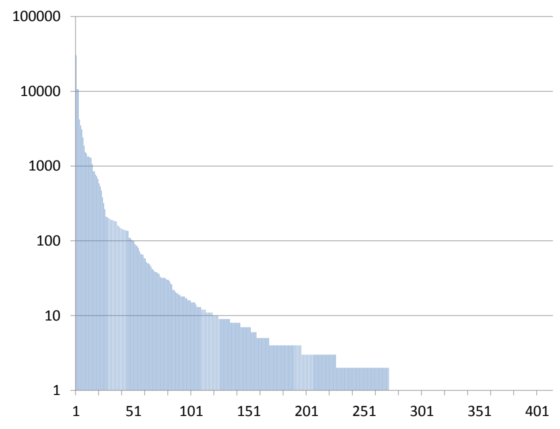

Figure 10: Sizes of clusters of samples with identical MAS footprints.

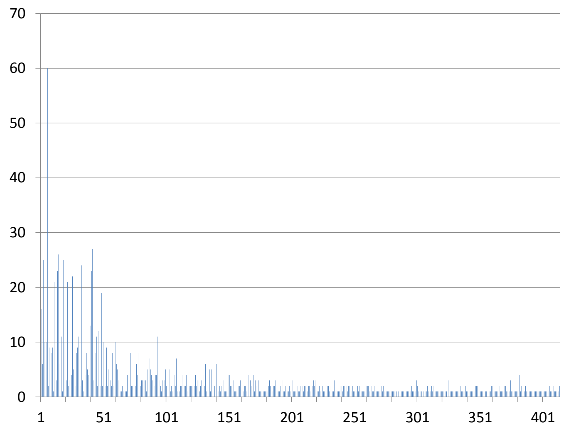

Figure 11: The numbers of different sized drivers loaded by samples of each cluster.

bytes that were modified. For hidden modules or processes, we simply used a tuple “HIDDEN_MODULE” or “HIDDEN_PROC”. We handled other behaviors similarly. Note that we carefully chose not to include any names or values that are easily modifiable by malware (e.g., a malicious driver’s name or a hidden module’s name). The tuples in each sample’s footprint are sorted so that we can easily compare two samples’ signatures.

We assigned samples into the same cluster if they had identical footprints. This mapped the 89,474 samples into 414 clusters whose sizes ranged from 1 to 30,411. A total of 272 clusters contained at least two samples. Figure 10 shows the distribution of cluster sizes.

To understand whether all samples in a cluster used a single kernel driver, we counted the number of different sized drivers loaded by samples in each cluster (see Figure 11). A total of 209 clusters have at least two different sized drivers loaded. This indicates that different malicious kernel drivers have shown identical MAS footprints. Thus we can potentially use MAS’s footprints to automatically detect new malware samples. We leave the investigation of this approach to future work.

9 Crash Dump Study

In this section we report our experience in using MAS to detect kernel rootkits in real-world crash dumps. Since the white list of trusted code is incomplete for the end user machines from which the crash dumps were collected, we cannot automate the process of rootkit detection entirely. However, we can leverage the findings from our kernel malware study to identify suspicious crash dumps before manually inspecting them.

From Table 2 we can see that the three most common behaviors of rootkits are hooking function pointers, hiding modules, and placing code hooks. Since many suspicious function pointers reported by MAS point to benign third-party drivers that are not on our white list, simply using the existence of suspicious function pointers is not an effective way to identify suspicious crash dumps. For rootkits that hook both function pointers and hide modules, the hooked function pointers usually do not point to a loaded module but either a pool block, a hidden module or some other memory region. We used this observation to ignore function pointers whose targets fall into loaded modules. We are aware that this may cause us to miss *non-stealthy* kernel malware that simply installs a driver. To handle such cases, we would need to either grow the white list or do more manual analysis. We also ignore function pointers whose targets do not appear to be the beginning of a function since they do not allow us to differentiate reliably between buggy rootkits and memory corruptions. In our study we used these conditions to do initial filtering to identify suspicious dumps. This initial filtering was done automatically.

For the eleven Windows Vista SP1 crash dumps, we found seven of them to be suspicious after the initial filtering. Our manual investigation confirmed that five crash dumps contain rootkits (e.g., hooking several driver’s dispatch routines, hiding its own driver). The other two were benign because the code hooks were placed by two anti-virus systems. Each of them hooked one of two very frequently called functions, *KiFastCallEntry* and *SwapContext*. We concluded that a code hook was placed by anti-virus software if the hook’s target falls into a module and internet search results indicated that the module belongs to an anti-virus vendor based on the module’s name.

For the 837 Windows 7 crash dumps, we found 177 suspicious dumps after the initial filtering. We quickly verified that 85 dumps that contain hidden modules were all infected by kernel rootkits. Out of the remaining 92 crash dumps, 82 dumps only contain code hooks, and the other ten contain suspicious function pointers that do not point to a loaded module. We manually analyzed these ten dumps and found that five of them contain rootkits and the other five have corrupted global function tables which let them pass the initial filtering. We cannot decide if the corruptions were due to a rootkit or a kernel bug. The 82 dumps with only code hooks have 37 different hooking patterns. For each hooking pattern, we picked one dump and manually inspected it with MAS's report. Surprisingly, all the code hooks appeared to be placed by anti-virus software.

In summary, with the process described above, we were able to quickly identify five Windows Vista SP1 dumps and 90 Windows 7 dumps that contain kernel rootkits. All the manual inspections described in this section took a total of less than one hour. This demonstrates that MAS is an effective tool for identifying rootkit footprints in real-world systems.

10 Related Work

MAS is not the first system that attempts to identify a kernel rootkit's footprint in a memory snapshot. But it is the first practical system that can do so with high accuracy, robustness and performance.

Our work was inspired by KOP [3]. While KOP is the first system to type dynamic data in a kernel memory snapshot with very high coverage, it lacks in robustness and performance. Our evaluation has shown that MAS is an order of magnitude faster than KOP in both static analysis and memory traversal. More importantly, when analyzing real-world crash dumps of systems running Windows Vista SP1, we observed no errors in MAS's output. In contrast, up to 40% of the function pointers reported by KOP appeared to be incorrect.

Kernel integrity checking has been studied in a large body of work. SBCFI [19] and Gibraltar [2] both leverage type definitions and manual annotations to traverse memory and inspect function pointers. Both fall short in data coverage as a result of not handling generic pointers [3]. A recent system called OSck [7] also discovers kernel rootkits by detecting modifications to kernel data. Instead of memory traversal, OSck identifies kernel data and their types by taking advantage of the slab allocation scheme used in Linux. It provides per-type allocations and enables direct identification of kernel data types. The slab allocator is unavailable on Windows operating systems, which makes Osck less useful for Windows. This problem cannot be solved by the mapping between pool tags and data types since it is *not* a one-to-one mapping. Worse, a pool tag may correspond to different types, and several data structures may be stored in one pool block.

MAS leverages source code and program-defined types to identify dynamic data and their types. Several other systems have tried to solve this problem without access to source code and type definitions. Laika [4] uses Bayesian unsupervised learning to infer data structures and their instances. REWARDs [11] recognizes dynamic data and their types when they are passed as parameters to known APIs at runtime. TIE [10] reverse engineers data type abstractions from binary programs based on type reconstruction theory and is not limited to a single execution trace. These reverse engineering tools are more effective for analyzing small to medium scale programs than for large-scale programs like the Windows kernel. Both MAS and KOP demonstrate that source code is critical for achieving high data coverage when analyzing kernel memory snapshots.

WhatsAt [20] is a tool for dynamic heap type inference. It uses type information embedded in debug symbols and attempts to assign a compatible program-defined type to each heap block by checking type constraints. If a block is untypable, WhatsAt uses it as a hint for heap corruptions and type safety violations. The main difference between WhatsAt and MAS is that whatsat cannot scale to large programs such as the Windows kernel.

MAS leverages a new demand-driven pointer analysis algorithm to enable precise but fast analysis for identifying type candidates for generic pointers in large-scale C/C++ programs. The key idea behind the demand-driven analysis is to formulate the pointer analysis problem as a Context-Free Language (CFL) reachability problem, which was explored in previous work [21, 24, 23, 27]. In [21], Reps first introduced the concept of transforming program analysis problems to graph-reachability problems. In [24], Sridharan *et. al.* apply this idea to demand-driven points-to analysis for Java. In [23], Sridharan and Bodik present a refinement-based algorithm for demand-driven context-sensitive analysis for Java. In [27], Zheng and Rugina describe a demand-driven alias analysis algorithm for C. We adopt their algorithm and extend it to support field-sensitivity. We also achieve context-sensitivity in a way similar to [23]. In KOP [3], Carbone *et. al.* extend the algorithm presented in [6] to be context- and field-sensitive. The key advantage of MAS over KOP is that MAS's static analysis can run in parallel.

MAS works on memory snapshots to analyze kernel rootkit behavior. Several other systems [9, 22, 26] have used virtualization-based dynamic tracing for the same purpose. Soft-timer based attacks [25] are detectable by MAS since the callback function pointer injected by the malware is always in memory and can potentially be detected by MAS.

11 Limitations

A key limitation faced by MAS is that an attacker who is familiar of MAS's design can potentially disrupt MAS's memory traversal by manipulating the kernel memory. MAS checks several constraints (see Section 4) before adding a new data object. If an attacker were able to find some pointer or enum fields in a data structure that may take arbitrary values without crashing the OS, he could potentially mislead MAS to reject instances of such a data structure by changing them to violate the pointer or enum constraints. The impact of this limitation remains unclear because we are not aware of such data structures. Moreover, even when such data structures exist, it is unclear if they will affect the identification of security sensitive data (e.g., hooked function pointers).

Another limitation of MAS is due to the existing implementation in Windows. Currently an attacker can modify the tag of a pool block without crashing Windows, and thus use it to mislead MAS. However, this limitation can be eliminated if the pool manager checks the tag of a pool block against the expected pool tag passed as a function argument when the pool block is freed.

12 Conclusions

We have presented MAS, a practical memory analysis system that can accurately and quickly identify a rootkit's memory footprint. We applied MAS to analyze 848 crash dumps collected from end user machines and 154,768 potential malware samples obtained from a major anti-virus vendor. Our experiments show that MAS was able to quickly analyze all memory snapshots with typical running times between 30 and 160 seconds per snapshot and with near perfect accuracy. With MAS, we were able to quickly identify 95 crash dumps that contain rootkits. Our kernel malware study shows that rootkits hooked 191 different function pointers in 31 different data structures. Furthermore, it demonstrates that many malware samples installed different kernel drivers but had identical memory footprints, which suggests a future research direction on leveraging memory footprints to automatically detect new malware samples.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. We are very grateful to many colleagues for their valuable feedback, suggestions and help throughout the effort of making MAS real: Alex Moshchuk, Anil Francis Thomas, Barry Bond, Bryan Parno, Chris Hawblitzel, David Molnar, Dennis Batchelder, Eddy Hsia, Galen Hunt, Gloria Mainar-Ruiz, Helen Wang, Jay Stokes, Jeffrey Chung, Jen-Lung Chiu, Jim Jernigan, Pat Winkler, Randy Treit, Reuben Olinsky, Rich Draves, Ryan Kivett, Scott Lambert, Tim Shoultz, YongKang Zhu.

References

[1] The Alureon rootkit. `http://en.wikipedia.org/wiki/Alureon`.

[2] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008).

[3] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (November 2009).

[4] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 255–266.

[5] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (October 2008).

[6] HEINTZE, N., AND TARDIEU, O. Ultra-fast aliasing analysis using CLA - a million lines of C code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation* (2001).

[7] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 279–290.

[8] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)* (October 2011).

[9] LANZI, A., SHARIF, M., AND LEE, W. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium* (2009).

[10] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (Feb. 2011), pp. 251–268.

[11] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)* (San Diego, CA, February 2010).

[12] MICROSOFT. Debug interface access SDK. `http://msdn.microsoft.com/en-us/library/x93ctkx8(VS.71).aspx`.

[13] MICROSOFT. Debugging Tools for Windows. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[14] MICROSOFT. PREfast. `http://msdn.microsoft.com/en-us/library/ff550543(v=vs.85).aspx`.

[15] MICROSOFT. Symbols and symbol files. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff558825(v=vs.85).aspx`.

[16] MICROSOFT. Windows driver kit. `http://msdn.microsoft.com/en-us/windows/hardware/gg487428.aspx`.

[17] MICROSOFT. Windows HPC Server 2008 R2. `http://www.microsoft.com/hpc`.

[18] MICROSOFT. Windows kernel pool tags. `http://msdn.microsoft.com/en-us/windows/hardware/gg463213.aspx`.

[19] NICK L. PETRONI, J., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (October 2007).

[20] POLISHCHUK, M., LIBLIT, B., AND SCHULZE, C. W. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 39–46.

[21] REPS, T. Program analysis via graph reachability. In *Proceedings of the 1997 International Logic Programming Symposium* (October 1997).

[22] RILEY, R., JIANG, X., AND XU, D. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM SIGOPS/EuroSys Conference on Computer Systems* (April 2009).

[23] SRIDHARAN, M., AND BODIK, R. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2006).

[24] SRIDHARAN, M., GOPAN, D., SHAN, L., AND BODIK, R. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems Languanges, and Applications (OOPSLA)* (October 2005).

[25] WEI, J., PAYNE, B. D., GIFFIN, J., AND PU, C. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008)* (December 2008).

[26] YIN, H., SONG, D., MANUEL, E., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)* (October 2007).

[27] ZHENG, X., AND RUGINA, R. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (January 2008).