Mapping Kernel Objects to Enable Systematic Integrity Checking

Martim Carbone
Georgia Institute of Technology
Atlanta, GA, USA
mcarbone@cc.gatech.edu

Weidong Cui
Microsoft Research
Redmond, WA, USA
wdcui@microsoft.com

Long Lu
Georgia Institute of Technology
Atlanta, GA, USA
long@cc.gatech.edu

Wenke Lee
Georgia Institute of Technology
Atlanta, GA, USA
wenke@cc.gatech.edu

Marcus Peinado
Microsoft Research
Redmond, WA, USA
marcuspe@microsoft.com

Xuxian Jiang
North Carolina State University
Raleigh, NC, USA
jiang@cs.ncsu.edu

ABSTRACT

Dynamic kernel data have become an attractive target for kernel-mode malware. However, previous solutions for checking kernel integrity either limit themselves to code and static data or can only inspect a fraction of dynamic data, resulting in limited protection. Our study shows that previous solutions may reach only 28% of the dynamic kernel data and thus may fail to identify function pointers manipulated by many kernel-mode malware.

To enable systematic kernel integrity checking, in this paper we present KOP, a system that can map dynamic kernel data with nearly complete coverage and nearly perfect accuracy. Unlike previous approaches, which ignore generic pointers, unions and dynamic arrays when locating dynamic kernel objects, KOP (1) applies interprocedural points-to analysis to compute all possible types for generic pointers (e.g., void*), (2) uses a pattern matching algorithm to resolve type ambiguities (e.g., unions), and (3) recognizes dynamic arrays by leveraging knowledge of kernel memory pool boundaries. We implemented a prototype of KOP and evaluated it on a Windows Vista SP1 system loaded with 63 kernel drivers. KOP was able to accurately map 99% of all the dynamic kernel data.

To demonstrate KOP's power, we developed two tools based on it to systematically identify malicious function pointers and uncover hidden kernel objects. Our tools correctly identified all malicious function pointers and all hidden objects from nine real-world kernel-mode malware samples as well as one created by ourselves, with no false alarms.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—*Program analysis*

General Terms

Security

CCS'09, November 9–13, 2009, Chicago, Illinois, USA.

Keywords

Kernel Integrity, Malware, Introspection, Pointer Analysis, Memory Analysis

1. INTRODUCTION

Modern operating systems are vulnerable to various types of attacks. In particular, kernel-mode malware represents a significant threat because of its ability to compromise the security of the kernel and, hence, the entire software stack. For example, kernel-mode malware can tamper with kernel code and data to hide itself and collect useful information from certain system events (e.g., keystrokes). To mitigate this security threat, the integrity of the complete kernel code and data needs to be verified.

It is relatively easy to check the integrity of kernel code and static data in a running system given their read-only nature and well-defined locations in memory. However, it is much harder to check the integrity of dynamic data due to their unpredictable memory locations and volatile nature. Not surprisingly, dynamic data have become one of the most attractive targets for kernel-mode malware [11, 12, 25].

Previous solutions for checking kernel integrity either limit themselves to kernel code and static data (e.g., system call tables) [13, 20, 24, 29], or can reach only a fraction of the dynamic kernel data [3, 22], resulting in limited security. Our study shows that previous systems [3,22] may miss up to 72% of the dynamic kernel data and thus may fail to identify function pointers manipulated by many kernel-mode malware. It is straightforward for an attacker, for instance, to implement new kernel-mode malware that tampers only with function pointers in objects that cannot be reached by these systems. Clearly, a *complete* and *accurate* understanding of all kernel memory is critical for checking the kernel's integrity.

Locating dynamic kernel objects in memory and identifying their types is the first and perhaps most difficult step towards enabling systematic integrity checks of dynamic kernel data. We call this process *mapping*. To locate a dynamic object, a *reference* to it must be found, usually in the form of a pointer. This pointer could, of course, be located in another dynamic object, turning this into a recursive problem. Mapping all the dynamic objects involves performing a complete traversal of the memory, starting from a set of globally well-defined objects and following each pointer reference to the next object, until all have been covered.

This basic idea was applied in previous security systems [3, 22]. However, these systems suffer from three major limitations. First, they cannot follow generic pointers (e.g., void*) because they only

leverage type definitions and thus cannot know the target types of generic pointers. Second, these systems cannot follow pointers defined inside unions since they cannot tell which union subtype should be considered. Third, they cannot recognize dynamic arrays and thus the objects inside them. Since generic pointers, unions, and dynamic arrays are programming paradigms commonly used in OS kernels, ignoring them may result in a very incomplete memory traversal as we observed in our study. Furthermore, previous systems require significant manual annotations in the source code. For example, in [3, 22], all linked list constructs needed to be annotated so that the corresponding objects can be correctly identified by the traversal. We have observed more than 1,500 doubly linked list types in the Windows Vista SP1 kernel. This large number makes an annotation process error-prone and time-consuming.

In this paper, we address the problem of automatically mapping all kernel objects to enable systematic kernel integrity checking. We present KOP (Kernel Object Pinpointer), a system that can map kernel data objects in a memory snapshot with nearly complete coverage and nearly perfect accuracy. Unlike previous systems, KOP is designed to address the challenges in pointer-based memory traversal. KOP's system architecture is shown in Figure 1. KOP first performs *static analysis* on the kernel's source code to construct an *extended type graph*. This extended type graph has not only type definitions and global variables but also all candidate target types for generic pointers. Given a memory snapshot, KOP then performs a *memory analysis* based on the extended type graph. KOP resolves type ambiguities caused by unions or generic pointers with multiple candidate target types and identifies dynamic arrays. The output is an *object graph* that contains all the identified kernel objects and their pointers to other objects. Systematic *kernel integrity checking* can be performed on this object graph.

KOP's ability to map kernel objects with high coverage and accuracy enables a variety of systematic kernel integrity checks. To concretely demonstrate the power of KOP, we have developed two tools based on it, namely, SFPD (for Subverted Function Pointer Detector) and GHOST (for General Hidden Object Scanning Tool). SFPD can systematically identify function pointers manipulated by kernel-mode malware. Compared with previous work such as SBCFI [22], SFPD's key advantages are: (1) it can check almost all function pointers due to the high coverage and accuracy of KOP's memory traversal, and (2) it can verify *implicit* function pointers—any function pointer field defined inside a union or not defined as a function pointer type (e.g., `unsigned int`) but sometimes used as a function pointer. GHOST is a tool that can systematically uncover hidden kernel objects. Unlike previous tools [21, 28] that rely on specific knowledge of a particular data type (e.g., process constructs), GHOST can work with arbitrary kinds of system objects without having to know how exactly they are organized in memory. Instead, GHOST can derive a view of certain system attributes (such as the list of active processes) from the kernel objects identified by KOP and compare it with information collected from an internal program. Mismatches in this comparison reveal hidden kernel objects.

We have implemented a prototype system of KOP using the Phoenix compiler framework [16] and evaluated it on a Windows system. Our system runs the Windows Vista SP1 kernel along with 63 drivers. Our experiments show that KOP's traversal reached 99% of all the dynamic kernel data. By verifying 94% of the mapped kernel data whose types were manually determined, we found that KOP correctly identified the types of 99%. We were not able to verify the other 6% simply because of the large number of different allocation contexts which we would have had to analyze manually. We also implemented a prototype system of SFPD and GHOST based on KOP. We tested SFPD with eight real-world kernel-mode rootkits. Our experiments show that SFPD identified all malicious function pointers, including implicit ones. We tested GHOST with two real-world kernel-mode rootkits and showed that it uncovered all the objects hidden by them. To compare KOP and its applications with previous approaches, we also conducted memory traversals in a way similar to the one described in [3, 22]. We found that previous approaches can only reach up to 28% of the dynamic kernel data and thus miss malicious function pointers for six of the eight rootkits we tested. A complete memory traversal with KOP currently takes eight minutes. This allows KOP to be used for a variety of offline applications. For instance, one can build SFPD-like tools based on KOP to analyze a memory snapshot taken from an infected machine for forensic analysis, or to profile memory footprints of kernel-mode malware for malware analysis.

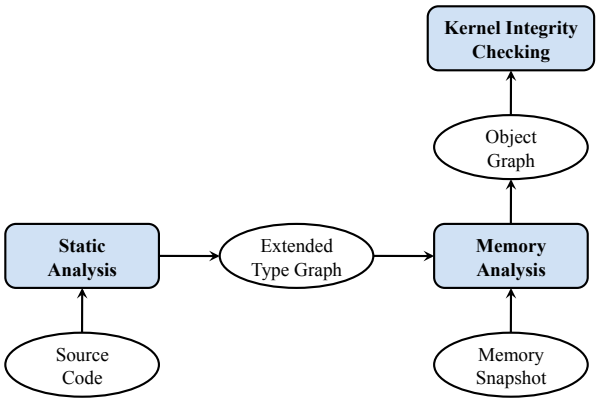

Figure 1: The KOP system architecture.

In summary, we make the following contributions:

- We designed a points-to analysis algorithm to perform an inter-procedural, field-sensitive, and context-sensitive analysis of a large C/C++ program such as an OS kernel (Section 3.1).
- We designed an algorithm to infer candidate types for generic pointers based on a points-to graph (Section 3.2).
- We designed a set of techniques to resolve type ambiguities in memory traversal (Section 4.1).
- We designed an approach to recognize dynamic arrays and their sizes in a memory snapshot (Section 4.2).
- We developed the first tool that can identify implicit function pointers manipulated by kernel-mode malware (Section 5).
- We developed a prototype system of KOP that can map dynamic kernel data in a Windows system loaded with a large number of drivers with nearly complete coverage and nearly perfect accuracy (Section 6).

2. OVERVIEW

The goal of KOP is to *completely* and *accurately* map all kernel objects in a memory snapshot in order to enable systematic kernel integrity checking. In KOP, we refer to a live instance of a data type (or, a data structure) as an *object*. KOP has two main components: a static analysis component and a memory analysis component.

```
1:  SLIST_ENTRY WrapDataListHead;

2:  typedef struct _WRAP_DATA {
3:    SLIST_ENTRY List;
4:    int32       Type;
5:    void*       PData;
6:  } WRAP_DATA;

7:  typedef struct _BIN_DATA {
8:    int32       BinLength;
9:    char*       BinData;
10: } BIN_DATA;

11: typedef struct _TXT_DATA {
12:   char*       TxtData;
13: } TXT_DATA;

14: void InsertSList
15: (SLIST_ENTRY *Head, SLIST_ENTRY *Entry)
16: {
17:   Entry->Flink = Head->Flink;
18:   Head->Flink = Entry;
19: }

20: void InsertWrapList (int32 type, void *data)
21: {
22:   WRAP_DATA *WrapData = AllocateWrapData();
23:   WrapData->Type = type;
24:   WrapData->PData = data;
25:   InsertSList(&WrapDataListHead, &WrapData->List);
26: }

27: void InsertTxtData(TXT_DATA *txt_data)
28: {
29:   InsertWrapList(0, txt_data);
30: }

31: void InsertBinData(BIN_DATA *bin_data)
32: {
33:   InsertWrapList(1, bin_data);
34: }
```

Figure 2: The source code for the running example.

KOP first performs static analysis on the kernel source code. It starts with an inter-procedural, inclusion-based points-to analysis [1] to derive a *points-to graph*. This is a directed graph whose nodes are pointers in the program and edges represent *inclusion* relationships. In other words, an edge from pointer x to pointer y means that any object pointers that can be derived from y are also derivable from x. Additionally, the points-to graph is maintained as a pre-transitive graph, i.e., the graph is not transitively closed [10].

Based on the pre-transitive points-to graph, KOP then infers candidate target types for generic pointers. *Generic pointers* are those whose target types cannot be extracted from their definitions. The term includes void* pointers as well as pointers defined inside linked list-related structures that are nested inside objects. The final output of KOP's static analysis component is an *extended type graph*. This is a directed graph where each node is either a data type or a global variable, and each edge has a label (m, n). This means that the pointer field at offset m in the source node points to the target node at offset n. We call this an *extended* type graph because it has edges corresponding to *generic pointer* fields which do not exist in the basic type graph derived from only type definitions.

Given a memory snapshot, KOP performs memory analysis by using the extended type graph to traverse the kernel memory. The output of the memory analysis component is an *object graph* whose nodes are instances of objects in the memory snapshot and edges are the pointers connecting these objects. Kernel data integrity checks can then be performed based on this object graph.

To help illustrate KOP, we will use the source code in Figure 2 as a running example. The code snippet shows the data structures and

```
_InsertWrapList:                                    #21
 _type, _data      = ENTERFUNCTION                  #21
 t282, {*CallTag}  = CALL* &_AllocateWrapData       #22
 _WrapData         = ASSIGN t282                    #22
 t283              = ADD _WrapData, 4               #23
 [t283]*           = ASSIGN _type                   #23
 t284              = ADD _WrapData, 8               #24
 [t284]*           = ASSIGN _data                   #24
 t285              = ADD _WrapData, 0               #25
 t286              = CONVERT t285                   #25
 CALL* &_InsertSList, &_WrapDataListHead, t286      #25
                      EXITFUNCTION                  #26
```

Figure 3: InsertWrapList() in medium-level intermediate representation (MIR).

functions for inserting a TXT_DATA object or a BIN_DATA object into a singly-linked list (WrapDataListHead). The list stores a group of WRAP_DATA objects.

3. STATIC ANALYSIS

KOP's static analysis component takes the kernel's source code as input, and outputs its extended type graph. To do so, we compute three sets of information: (1) object type definitions, (2) declared types and relative addresses of global variables, and (3) candidate target types for generic pointers. Since it is straightforward to retrieve the first two sets of information from a compiler, we will focus on how the candidate target types for generic pointers are determined. We first describe how we perform an inter-procedural points-to analysis [1] to construct a points-to graph. We then describe how we derive target types for generic pointers based on the points-to graph and the type definitions of local and global variables. Our static analysis is based on the medium-level intermediate representation (MIR) used by the Phoenix compiler framework [16]. In Figure 3, we show the MIR for the function InsertWrapList() of our running example.

3.1 Points-To Analysis

Our inter-procedural flow-insensitive (i.e., ignoring the control flow within a procedure) points-to analysis is due to Andersen [1]. It computes the set of logical objects that each pointer may point to (referred to as the points-to set for that pointer). The logical objects include local and global variables as well as dynamically allocated objects. Since our goal is to find candidate target types for generic pointers, our points-to analysis must be *field-sensitive* (i.e., distinguishing the fields inside an object). Furthermore, to achieve good precision, we chose to perform *context-sensitive* analysis (i.e., distinguishing the calling contexts). The reason is that generic functions such as InsertSList from our running example are widely used in OS kernels, and without context-sensitivity, the analysis of such functions would result in very general points-to sets for their arguments. Basically, all list heads and entries that are ever passed to such a generic function would point to each other. Finally, our points-to analysis must scale to a large codebase such as an OS kernel.

Points-to analysis for C programs has been widely studied in the programming languages field [2, 6, 9, 10, 19, 30, 31]. Unfortunately, none of the previous algorithms meets our requirements. All the previous solutions chose to sacrifice precision for performance since the points-to analysis used inside compilers is expected to finish within minutes. When designing KOP, we decided to *revise the algorithm proposed by Heintze and Tardieu in [10] to achieve field-sensitivity and context-sensitivity*. Note that the original algorithm is context-insensitive and field-based. In field-based analysis, all instances of a field are treated as one variable, whereas in

Rule	Original	KOP
Assign	$x = y \Longrightarrow \langle x, y\rangle$	$(x = y + n, op) \Longrightarrow \langle x, y, n, op\rangle$
Trans	$\langle x, y\rangle, \langle y, z\rangle \Longrightarrow \langle x, z\rangle$	$\langle x, y, n_1, ops_1\rangle, \langle y, z, n_2, ops_2\rangle \Longrightarrow \langle x, z, n_1 + n_2, ops_2 + ops_1\rangle$ where $ops_1 + ops_2$ is a valid call path.
Star-1	$\langle x, \&z\rangle, *x = y \Longrightarrow \langle z, y\rangle$	$\langle x, \&z, n, ops\rangle, (*x = y, op) \Longrightarrow \langle z.n, y, 0, op + rev(ops)\rangle$ where $op + rev(ops)$ is a valid call path.
Star-2	$\langle y, \&z\rangle, x = *y \Longrightarrow \langle x, z\rangle$	$\langle y, \&z, n, ops\rangle, (x = *y, op) \Longrightarrow \langle x, z.n, 0, ops + op\rangle$ where $ops + op$ is a valid call path.

Table 1: Deduction rules used by the original algorithm [10] and KOP.

field-sensitive analysis, each instance is treated separately. Consequently, field-sensitive analysis is more precise.

Next we describe in detail how we achieve field-sensitivity and context-sensitivity in our points-to analysis. We will focus on the changes introduced to Heintze and Tardieu's algorithm [10].

By using temporary variables, Heintze and Tardieu transform pointer assignments into four canonical forms: $x = y$, $x = \&y$, $*x = y$, and $x = *y$. To handle pointer offsets, we generalize the first two assignment forms to $x = y + n$ and $x = \&y + n$ where n is a pointer offset. To achieve context-sensitivity, we associate each assignment with a variable *op* that specifies the *call* or *return* operation involved in the assignment. *op* is null when the assignment occurs inside a single function.

In [10], given the four canonical assignment forms, an edge in the points-to graph is a pair $\langle src, dst\rangle$. Four deduction rules are used to compute the points-to graph (shown in the left portion of Table 1). To consider pointer offsets and calling context changes, we add a label $\langle n, ops\rangle$ to each edge. We denote a labeled edge from src to dst by $\langle src, dst, n, ops\rangle$. For example, given the pointer assignment $\_Entry = t286$ due to the function call at line 25 of Figure 3, the corresponding edge will be $\langle \_Entry, t286, 0, call@file : 25\rangle$.

Given the edge labels, we change the deduction rules accordingly (shown in the right portion of Table 1). The changes related to field-sensitivity are straightforward. In the **Assign** rule, the pointer offset is simply included in the edge's four-tuple. In the **Trans** rule, the pointer offsets are added up. In the **Star** rules, we create a new node $z.n$ to represent an instance of the pointer field at offset n in logical object z to achieve field-sensitivity. In our deduction rules, whenever we create a new edge, we also check if the sequence of call/return operations involved is valid under context-sensitivity. A sequence is valid if it can be instantiated from a valid call path (i.e., a control flow). We assume there are no recursive functions (we have not observed any in the Windows source code we analyzed). So a valid call path has at most a single call at each call site. Additionally, we do not need to apply any special rules to global variables since we create a single node for each global variable disregarding the function contexts. This allows information to flow through global variables between different functions.

To avoid the cost of computing the full transitive closure, Heintze and Tardieu maintain a pre-transitive graph and compute the points-to set on-demand. We adapt their algorithm to take our edge labels into account. Compared with the original algorithm, our pre-transitive graph algorithm has two key differences. First, we enforce context-sensitivity by checking if a sequence of call/return operations is *valid*. Second, whenever a cycle is found, the algorithm in [10] merges all the nodes in the cycle. Instead we terminate the path traversal in this case, because our edges carry more information than just pointer inclusions. The cycle detection in our pre-transitive graph algorithm and the no-recursive-call policy in enforcing context-sensitivity ensure that our points-to analysis terminates.

3.2 Inferring Types for Generic Pointers

The output of our points-to analysis is a pre-transitive points-to graph from which we can derive the candidate target types for generic pointers. The key idea is to leverage the type definitions of local and global variables. Before describing our algorithm in detail, we will use an example to explain the intuition behind it.

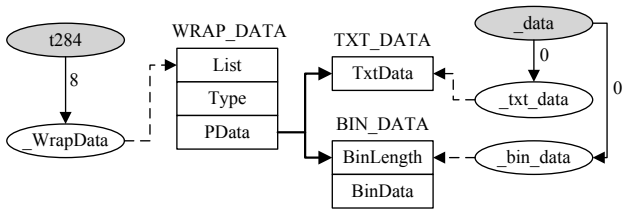

Figure 4: An example for inferring candidate target types of generic pointers. In this example, we derive the types for WRAP_DATA.PData **from the assignment** *t284 = _data **(see the MIR code in Figure 3). This graph is a mix of the points-to graph and the extended type graph. It illustrates how we derive edges in the extended type graph based on the points-to graph. Ellipse nodes and solid arrows are part of the points-to graph. Rectangular nodes and bold-solid arrows are part of the final extended type graph. The dashed arrows are derived from the type definitions of variables.**

The basic idea of our algorithm is illustrated in Figure 4. In the points-to graph of our running example, we have edges from t284 to _WrapData (with pointer offset 8) and from _data to _bin_data and _txt_data (with pointer offset 0). In addition, based on the type definitions, we know that _WrapData points to WRAP_DATA, _bin_data points to BIN_DATA and _txt_data points to TXT_DATA. Then, given the assignment *t284 = _data, we can infer that WRAP_DATA+8, which is WRAP_DATA.PData, may point to either BIN_DATA or TXT_DATA. The key difference here from classic points-to analysis is that, *although a pointer like* _WrapData *may not point to any log-*

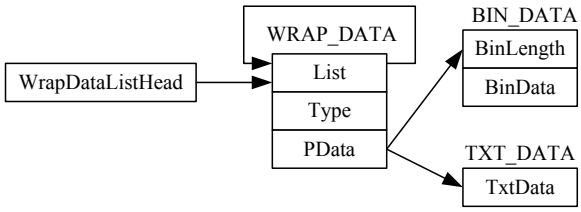

Figure 5: The extended type graph for the running example.

ical object, we leverage its type definition to derive the target types for WRAP_DATA.PData. Moreover, with the pointer offsets from the points-to graph, we naturally identify that WRAP_DATA.List does not just point to an SLIST_ENTRY object but actually a WRAP_DATA object. With this, KOP avoids the need for manual annotations in the code for types such as SLIST_ENTRY. The extended type graph for our running example is shown in Figure 5. Note that WrapDataListHead is a global variable and the other nodes are data types.

More specifically, for each assignment in the form $*x = y$, we first search for all the reachable nodes in the pre-transitive pointer graph for x and y, separately. We refer to them as $TargetSet(x)$ and $TargetSet(y)$. Then for each node a in $TargetSet(x)$ and each node b in $TargetSet(y)$, we check if there is a valid call path from a to b. If there is one, we derive a candidate target type for the corresponding pointer field in a's data type. Similarly, we derive candidate types from assignments in the form $x = *y$. The intuition is that, when y is a generic pointer such as void*, it will be cast back to its actual type before the program accesses the data pointed to by it. Specifically, for each assignment, we first search for the nodes that can reach x, referred to as $SourceSet(x)$. Then for each node a in $SourceSet(x)$ and each node b in $TargetSet(y)$, we check if there is a valid call path from a to b. If so, we derive a candidate type for the corresponding pointer field in a's data type.

A problem inherent to flow-insensitive points-to analysis is its imprecision. To mitigate this problem, we introduce a constraint when deriving candidate types for generic pointers in linked list constructs. For example, a pointer field in SLIST_ENTRY must point to an SLIST_ENTRY structure. This kind of constraint reduces the number of incorrect candidate target types and thus reduces the possibility of errors in the memory analysis. Such constraints do not decrease KOP's coverage because all valid candidate target types are expected to meet this constraint.

4. MEMORY ANALYSIS

KOP's memory analysis component maps kernel data objects and derives the object graph for a given memory snapshot. It does so by using the extended type graph derived earlier to traverse the kernel memory. We use our running example to explain the basic idea behind our memory traversal algorithm and the challenges we faced.

Starting at global variable WrapDataListHead, KOP first reaches an object of type WRAP_DATA referenced by it. KOP then follows each pointer field defined inside this object. By following the field WRAP_DATA.List (a linked list structure), KOP reaches another object of type WRAP_DATA, and continues by following each pointer field inside it. This technique has its roots in trace-based garbage collection, where it is used to identify all referenced memory blocks in a program's heap.

A challenge arises when trying to follow the pointer field WRAP_DATA.PData. This field is a generic pointer which, according to the extended type graph, can either point to a BIN_DATA object or a TXT_DATA object. KOP must determine the type of the object referenced by WRAP_DATA.PData in memory. Additionally, the BIN_DATA.BinData and TXT_DATA.TxtData could be pointers to dynamic arrays. Finally, KOP needs to tolerate identification errors to a certain degree.

In summary, to correctly identify kernel objects, KOP faces three challenges: resolving type ambiguities, recognizing dynamic arrays, and controlling identification errors. In the rest of this section, we describe in detail how we address each of these challenges. We use examples from the Windows operating system but our techniques are applicable to other operating systems (e.g., Linux) since they rely on common implementation paradigms used in modern operating systems.

4.1 Resolving Type Ambiguities

Type ambiguities come from two sources: unions and generic pointers that have multiple candidate target types. We will refer to the range of possible choices in both cases as *candidate types* or *candidates*. KOP is the first system that can resolve type ambiguities in memory traversal.

KOP considers two constraints when determining the correct candidate type. The first is a size constraint. Specifically, operating system kernels (e.g., Windows) store dynamic kernel data in a set of memory allocation units called *pool blocks*. Each pool block is created by a call to a memory allocation function (e.g., ExAllocatePool() in Windows). Each kernel object must lie completely within a single pool block. We consider this as a hard constraint. When resolving type ambiguities, KOP rejects any candidate that violates the size constraint.

The second constraint is based on the observation that the data stored by certain data types must have specific properties. Currently, we apply this constraint only to pointer fields. With certain exceptions, pointer fields in kernel objects are either null or assume values in the kernel virtual address range (e.g., `[0x80000000, 0xFFFFFFFF]` for 32-bit Windows). Drivers that directly access user mode memory, for instance, do not meet this condition. Thus, we treat it as a soft constraint. We accept candidates that violate this constraint as long as the number of violating pointers is sufficiently small. More precisely, given several candidate types, we compute for each candidate the fraction of pointer fields that violate the constraint and choose the one with the lowest fraction. We discard the candidate if the fraction of invalid pointer values for it is too high (e.g., >10%).

These two constraints are not only evaluated on the candidates themselves, but also recursively for their "child" objects (i.e., the objects pointed by the candidates) up to a certain depth level (e.g., three). By doing so, we improve the accuracy of type ambiguity resolution since we have more data to rely upon when making the decision.

4.2 Recognizing Dynamic Arrays

Dynamic arrays are widely used in OS kernels and drivers. KOP is the first system with the capability to automatically recognize dynamic arrays in memory traversal. The key idea is to leverage the kernel memory pool boundaries, i.e., a dynamic array must fit into a single pool block. Moreover, we note that a dynamic array is usually allocated in two possible ways: it may take up a whole pool block, or it may extend an object whose last field is defined as an array of size 0 or 1. Based on these two observations, KOP checks each allocated pool block to recognize dynamic arrays after the object traversal (without dynamic arrays) is completed.

If a single object is identified at the start of a pool block, KOP

analyzes the block further to determine if it contains a dynamic array of the first kind. The intuition is that arrays are typically accessed via a pointer to their first element. KOP then tests if the array candidate meets a new size constraint: the size of a pool block must be a multiple of the size of the first object plus some number between 0 and $A-1$, where A is the pool block alignment. This is a hard constraint. Finally, KOP checks the pointer value constraint for each array element. KOP accepts the dynamic array candidate if a sufficiently large fraction of array elements (e.g., >80%) have a low fraction of invalid pointer values.

KOP checks a pool block for a dynamic array of the second kind if there is an empty space (i.e., no objects were found) trailing an object and the object's last element is an array of size 0 or 1. For such objects, KOP checks the size and pointer value constraints as described above.

After identifying dynamic arrays, KOP uses them as roots and reruns the traversal algorithm. This process is repeated until no more dynamic arrays can be found.

4.3 Controlling Object Identification Errors

During the memory traversal, KOP may incorrectly identify an object for three main reasons: (1) choosing the wrong candidate when resolving type ambiguities, (2) mistaking a dynamic array, and (3) program bugs (e.g., dangling pointers). Given the recursive nature of KOP's memory traversal, an incorrect object may cause more errors during the rest of the traversal. Therefore, it is critical to reduce identification errors and prevent them from propagating. To do so, we employ the following two techniques.

First, instead of performing a single complete traversal, KOP traverses the kernel memory in multiple rounds. The key idea is to *identify unambiguous kernel objects and use them to constrain the solution space.*. Specifically, KOP performs the memory traversal in three distinct rounds. In the first round, KOP identifies all the global objects and those objects referenced by global pointers. These are the roots used in the traversal and are likely to be correct. In the second round, starting from the objects found in the first round, KOP traverses the kernel memory *but only follows pointer fields that do not have type ambiguities*. We do not infer dynamic arrays in this round either. This way we avoid the identification errors that may be caused by either resolving type ambiguities or inferring dynamic arrays. In the third round, starting from the objects found in the previous rounds, KOP traverses the kernel memory and resolve type ambiguities when necessary. KOP also identifies and traverses dynamic arrays in this round (after the traversal without dynamic arrays is finished). Note that, if two objects identified in the same round conflict with each other, we keep both of them. Currently, we perform a depth-first traversal in each round.

Second, to limit the damage caused by an earlier identification error, KOP uses a safe-guard mechanism. Whenever following a typed pointer during the traversal, KOP first checks if the object implied by the pointer type meets the constraints used to resolve type ambiguities (see Section 4.1). This can be treated as a special case in which only a single candidate is considered. If the object violates either constraint, KOP discards it and stops that branch of the traversal.

5. KERNEL INTEGRITY CHECKING

We implemented two integrity checking applications on top of KOP: function pointer checking and hidden object discovery. We chose these applications because they address two of the most common techniques used by kernel-mode malware, especially rootkits.

5.1 Function Pointer Checking

Function pointers are commonly used throughout the kernel to perform indirect calls. A popular technique used by malware is to change their values to point to malicious code, an action also known as *hooking*. By doing so, malware can hijack the OS control flow whenever an indirect call of these function pointers occurs. This allows it to intercept and control certain types of system activity.

A common task in detecting unknown or analyzing known kernel-mode malware is to identify all the function pointers manipulated by the malware. The ideal way to do this is to inspect the values of all function pointers in the kernel and determine if they point to legitimate targets. There are several difficulties with this. First, many function pointers reside in dynamic kernel objects, and therefore do not have a fixed location in memory. Second, inside a single object, not all function pointers can be unequivocally identified. This can happen in the following two scenarios: (1) a field is not declared as a function pointer type (e.g., `unsigned int`) but effectively used as a function pointer, and (2) a function pointer is defined inside a union. We refer to these as *implicit* function pointers and all the others as *explicit* function pointers. Thus, the task of complete and accurate function pointer identification is a challenge in modern OSes.

To address these problems we built SFPD, the *Subverted Function Pointer Detector*. SFPD relies on KOP to perform a systematic analysis of function pointers in a kernel memory snapshot. Particularly, it leverages KOP's nearly complete memory traversal to identify kernel objects. Due to KOP's greater coverage of the kernel memory, SFPD is able to verify the function pointers of a much larger set of objects than previous approaches such as SBCFI [22]. SFPD also leverages KOP's points-to analysis to recognize implicit function pointers. SFPD is the first system that can identify malicious implicit function pointers in kernel memory.

SFPD is given a white list of trusted modules. This includes the kernel and trusted drivers. Given a memory snapshot, SFPD first checks if the code of these modules was modified. If so, any modified parts of the code are marked as untrusted. The rest of the code is treated as trusted. SFPD then checks every function pointer in the kernel objects found by KOP based on the following policy: *An explicit function pointer must point to trusted code; an implicit function pointer must point to either trusted code or a data object found by KOP; otherwise, the function pointer is marked as malicious*.

This policy is simple but powerful. For example, SFPD can detect any function pointer that targets untrusted code placed in unused blocks of memory. At the same time, by leveraging KOP's high coverage, it effectively avoids the false alarms that would otherwise be caused in two cases: (1) Our flow-insensitive points-to analysis mistakenly identifies data pointers as implicit function pointers, due to imprecision; and (2) data pointers share the same offset as a function pointer in a union.

Additionally, we leverage the traversal information generated by KOP to retrieve the traversal path to objects whose function pointers were found to be malicious. Such information is important because this path often reveals the purpose of the function pointer. For instance, simply knowing about a function pointer in an `EX_CALLBACK_ROUTINE_BLOCK` object [4] does not tell us what it is for. We will, however, know that it is used to intercept process creation events when SFPD shows that it is referenced from a global pointer array in `PspCreateProcessNotifyRoutine` [4, 27].

5.2 Hidden Object Discovery

A technique often employed by kernel-mode malware is to hide itself by either hijacking control flow or directly manipulating ker-

nel objects. For instance, to hide a process in the Task Manager, an attacker can either hijack the system call to NtQuerySystemInformation or unlink the corresponding process object from the active process list. Previous efforts have focused on detecting specific types of hidden objects by hardcoding expert knowledge of the related data structures [21,28]. Such approaches are time-consuming, and require a human expert with deep knowledge of the system to create the rules.

Given KOP's ability to map kernel objects, we developed a tool called *General Hidden Object Scanning Tool* (GHOST) that can systematically uncover hidden objects of arbitrary type with little human effort. Specifically, given an object type, GHOST compares the list of all the objects of that type found by KOP in a memory snapshot with the list of objects returned by a program such as Task Manager. One may need to repeat this comparison multiple times to avoid false alarms caused by state variations from the time that the internal program is executed to the time that the memory snapshot is taken. Currently, GHOST uses the information reported by Task Manager and WinObj [26] and compares it with the data returned by KOP to uncover hidden processes and drivers.

Compared with previous approaches, GHOST has two key advantages. First, the amount of manual effort is small since the deep knowledge of data structures resides inside KOP. For instance, to get the list of loaded drivers using KOP, one just needs to know that a specific pointer in each driver object refers to the driver's name. Second, KOP's exhaustive traversal of all the pointer paths allows GHOST to identify a hidden kernel object as long as there exists at least one pointer path to it.

6. IMPLEMENTATION AND EVALUATION

We developed a prototype of KOP on Windows. The static analysis component was built using the Phoenix compiler framework [16]. The runtime component is a standalone program. Both components were implemented in C# with a total of 16,000 lines of code. KOP operates in an offline manner on a snapshot of the kernel memory, captured in Windows as a complete memory dump [15]. KOP relies on the Windows Debugger API [14] to resolve symbols, access virtual addresses, and extract information about the pool blocks allocated in the snapshot.

We used the Windows Vista SP1 operating system as our analysis subject. Its kernel and drivers are mostly written in C, with parts in C++ and assembly. Our experiments were performed on a system loaded with 63 kernel drivers shipped with the OS. We ran this system in a VMware virtual machine with 1GB RAM. In our prototype, we used the following parameters for the memory analysis: tolerance of at most 10% for invalid pointer values in an object, requirement of at least 80% of the dynamic array elements to meet the pointer constraint, and the use of three levels of child objects when evaluating the pointer constraint for a candidate.

Several implementation techniques in the Vista kernel and drivers presented difficulties for KOP. We were able to identify the following cases: (1) the lower bits in some pointers are used to store a reference counter (assuming that the target is 8-byte aligned) (2) in some cases relative memory offsets are used for object referencing, and (3) several cases of implicit type polymorphism in C (e.g., a single object can be used as if it belonged to more than one type). In developing our prototype, we manually adjusted our implementation to handle these cases.

We also implemented prototype systems for SFPD and GHOST. Our SFPD prototype itself has a total of 1,000 lines of C# code, and our GHOST prototype has 200 lines of C# code. The relatively small size of our SFPD and GHOST prototypes shows that, given the infrastructure provided by KOP, it requires only a small amount of extra effort to implement an integrity checking application. In the rest of this section, we present the evaluations of KOP, SFPD, and GHOST.

6.1 KOP

KOP's main goal is to completely and accurately map the kernel objects in a memory snapshot. Since we can trivially identify all static kernel objects by mapping global variables, we will only evaluate KOP's *coverage* of dynamic kernel objects. We also evaluated KOP's *performance* to demonstrate that it can perform its offline memory analysis in a reasonable amount of time. Before presenting our experimental results on coverage and performance, we will first summarize the results of our static analysis.

6.1.1 Static Analysis

We applied KOP's static analysis to the source code of the Vista SP1 kernel and the 63 drivers, with a total of 5 million lines of code. This codebase contains 24423 data types and 9629 global variable definitions. KOP derived the candidate target types for 3228 void* pointers, 1560 doubly linked lists, 118 singly linked lists, and 8 triply linked lists (i.e., balanced trees). KOP also identified 3412 implicit function pointers. In our experiments, KOP needed less than 48 hours to complete its static analysis on a 2.2GHz Quad-Core AMD Opteron machine with 32GB RAM. Since KOP only needs to run its static analysis once for an OS kernel and its drivers, we consider this running time acceptable.

6.1.2 Coverage

We measured KOP's coverage by the fraction of the total allocated dynamic kernel memory for which KOP was able to identify the *correct* object type. Ideally, we would use a ground truth that specifies the exact object layout in kernel memory. However, obtaining such a ground truth is extremely difficult and time-consuming. For instance, the value of a certain field in an object may implicitly determine the existence and layout of other objects in the same pool block. Thus, we would need to understand the semantics of each object field to obtain the exact object layout.

Instead, we obtained a ground truth with a slightly coarser granularity. Specifically, we instrumented the kernel to log every pool allocation and deallocation during runtime, along with the call stack, address and size.

We manually inspected the source code for each location on the call stack. This allowed us to identify a call stack location at which the types of the allocated objects could be readily identified in the source code. This was often not the stack location at which the generic allocation function (ExAllocatePool()) was called, but some other location higher in the call stack. We manually analyzed 367 allocation sites and identified the object types that can be allocated at each site. This corresponds to 95% of the allocated pool blocks (94% of the allocated bytes). We were not able to do this for 100% of the pool blocks simply because of the very large number of different allocation sites for the remaining 5%.

Since our ground truth does not specify the exact object layout, we do not know the exact number of objects that exist in the pool blocks. Therefore, we cannot measure KOP's coverage based on the fraction of correctly identified objects. Instead, we measured the coverage based on bytes, since we know the total number of bytes in allocated pool blocks.

For a byte b inside a pool block that is part of our ground truth, we say b is *correctly mapped* if KOP identified a single object which contains b's location and, under our ground truth, the object type is contained in the pool block. If b is mapped to an object of incorrect type or more than one type, we say it was *incorrectly*

	Clean-Boot (%) – Total bytes: 42775648							Stress-Test (%) – Total bytes: 50588704						
Type	CM	IM	MG	UM	MOG	VC	GC	CM	IM	MG	UM	MOG	VC	GC
Basic	25.4	0.0	68.9	1.4	4.3	**26.9**	**26.8**	26.6	0.0	68.0	1.4	4.0	**28.1**	**28.0**
KOP	93.7	0.0	0.6	5.3	0.4	**99.3**	**98.9**	93.8	0.0	0.8	5.0	0.4	**99.2**	**98.8**

Table 2: Coverage results for the basic traversal and KOP when applied to the clean-boot and stress-test memory snapshots. CM = Correctly Mapped, IM = Incorrectly Mapped, UM = Unverified Map, MG = Missed in Ground-truth, MOG = Missed Outside Ground-truth, VC = Verified Coverage and GC = Gross Coverage. The numbers in the table are percentages of the total number of bytes.

mapped by KOP. Finally, if it was not mapped at all, we say it was *missed under ground-truth*. Let CM, IM and MG be the sets of bytes that are classified as correctly mapped, incorrectly mapped and missed under ground-truth, respectively. We define *verified coverage* as

$$\frac{|CM|}{|CM| + |IM| + |MG|} \quad ,$$

where $|\cdot|$ denotes the set size. We chose the allocation sites for which we computed the ground truth based only on the number of pool blocks they cover and not based on any properties of KOP. Therefore, we believe that the verified coverage has the character of a statistical sample and that it is representative of KOP's overall coverage.

To gain further confidence, we compute a second measure of coverage. Consider any byte b in a pool block that is not in our ground truth. We say that b is an *unverified mapping* if KOP identified some object at its location and *missed outside of ground-truth* otherwise. Let UM and MOG denote the respective sets. We define *gross coverage* as

$$\frac{|CM| + |UM|}{|CM| + |IM| + |MG| + |UM| + |MOG|} \quad .$$

In our coverage experiments, we compared KOP with a *basic* traversal algorithm. Like previous approaches [3, 22], the basic traversal follows only typed pointers and doubly linked lists without resolving type ambiguities and recognizing dynamic arrays. The only difference is that our basic traversal algorithm uses the target types of linked lists automatically derived from KOP's static analysis, while previous approaches relied on manual efforts. To demonstrate KOP's robustness with different workloads, we tested it on two different memory snapshots. One was collected right after the system was booted up, and the other was collected after running a large number of system and user processes on the system for 15 minutes. We refer to these two memory snapshots as the *clean-boot* and *stress-test* snapshot.

The experimental results for the coverage of KOP and the basic traversal algorithm are shown in Table 2. The total size of the dynamic kernel data is 42.7MB in the clean-boot memory snapshot and 50.6MB in the stress-test snapshot. In both snapshots, KOP's verified coverage and gross coverage are 99%, whereas for the basic traversal it is only 28%. Since our ground truth covers 94% of the dynamic kernel data, the gross coverage is very close to the verified coverage, as shown in Table 2. We manually investigated some of the cases where KOP either identified the objects incorrectly or missed them completely. We found that they were due to three reasons: KOP incorrectly resolving type ambiguities or recognizing dynamic arrays, dangling pointers and unorthodox Windows kernel implementation techniques that we were not able to identify. In Section 8, we will discuss future research directions that can help mitigate these errors.

6.1.3 Performance

We measured KOP's running time when analyzing twelve distinct memory snapshots used in our experiments (including those used on SFPD's and GHOST's evaluations). We used a 4GHz Intel Xeon Duo Core machine with 3GB RAM. The median running time was 8 minutes, including the overhead of reading the memory snapshot stored on the disk. We consider this running time to be acceptable for offline analysis.

6.2 SFPD

The goal of SFPD is to identify all malicious function pointers in the kernel memory. We evaluated SFPD by analyzing the memory snapshots of systems infected with kernel-mode malware. Specifically, given a malware sample, we executed it in the Windows Vista SP1 virtual machine we used to evaluate KOP, and then generated a memory snapshot after waiting for a few seconds.

For each memory snapshot, we manually built the ground truth of all malicious function pointers. More precisely, we first manually identified the code regions occupied by the malware based on our instrumentation logs. We then conducted an exhaustive memory search for memory locations pointing to the regions containing the malware's code. We then manually verified each of them to check if they were malicious function pointers.

In our experiments, we tested SFPD with eight real-world kernel malware samples collected from a public database [17]. Running on a 4GHz Intel Xeon Duo Core machine with 3GB RAM, SFPD finishes a scan of a memory snapshot in less than two minutes, excluding the time KOP takes to map kernel objects in the snapshot.

Our experimental results for SFPD are shown in Table 3. We do not report results on the System Service Dispatch Tables (SSDTs) and the Interrupt Dispatch Table (IDT) hooks because these are static data and therefore not our focus. We compared SFPD with a baseline algorithm which is similar to previous approaches [3, 22]. This baseline algorithm inspects explicit function pointers based on the kernel objects identified by the basic traversal. SFPD identified all the malicious function pointers for all eight malware samples with *zero* false alarms. However, the baseline algorithm missed malicious explicit function pointers placed by seven of the eight malware samples, as well as *all* of the implicit function pointers. This was a result of the basic traversal's low memory coverage, as well as its lack of knowledge of implicit function pointers. For instance, the basic traversal fails to identify the EX_CALLBACK_ROUTINE_BLOCK object added by the malware because it is referenced by the global variable PspCreateProcessNotifyRoutine via a generic pointer.

The baseline algorithm is able to *detect the existence* of all the eight real-world malware samples we tested. After all, to determine that a system is infected, it is enough to identify just one malicious function pointer (including entries in SSDTs or IDT not shown in Table 3). However, it is straightforward to create a new rootkit that only tampers with function pointers missed by the baseline algorithm. For instance, a rootkit can hook an EX_CALLBACK_ROUTINE

Name	Malicious function pointer	Type	Baseline	SFPD
Trojan.Dropper.Farfli.G	DRIVER_OBJECT.DriverInit	E	**0**/2	2/2
	DRIVER_OBJECT.MajorFunction[]	E	**0**/30	30/30
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/2	2/2
	ETHREAD.Win32StartAddress	I	**0**/2	2/2
VirTool: WinNT/Syspro.A	DRIVER_OBJECT.DriverInit	E	1/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	28/28	28/28
	FAST_IO_DISPATCH.*	E	21/21	21/21
	FS_FILTER_CALLBACKS.*	E	12/12	12/12
	NOTIFICATION_PACKET.NotificationRoutine	E	1/1	1/1
TrojanDropper: Win32/Cutwail.K	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	**2**/6	6/6
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
VirTool: WinNT/Odsrootkit.C	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.DriverUnload	E	**0**/1	1/1
Backdoor: WinNT/Syzor.A	DRIVER_OBJECT.MajorFunction[]	E	4/4	4/4
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
Rootkit.Win32.Agent.fwz	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	1/1	1/1
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
Trojan: Win32/DriverByPass	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.DriverUnload	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	4/4	4/4
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	KAPC.KernelRoutine	E	6/6	6/6
Backdoor: Win32/Haxdoor	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	**0**/2	2/2

Table 3: Results from applying SFPD to eight memory snapshots infected with different real-world malware samples. Function pointers are classified as either *explicit* (E) or *implicit* (I) based on their kind. A/B means that a scheme detects A out of B malicious function pointers. Definitions of the data structures are available as part of the Windows Research Kernel [4].

_BLOCK object pointed by PspCreateProcessNotifyRoutine so that its code is executed whenever a process is created. Furthermore, all previous approaches including the baseline algorithm do not inspect implicit function pointers. Therefore they will miss rootkits that only hook such function pointers. SFPD, on the other hand, is able to identify subverted implicit function pointers by leveraging KOP's static analysis and near complete memory coverage. In Table 3, we can see that SFPD successfully identified all the malicious implicit function pointers.

Moreover, KOP's high coverage is critical to prevent SFPD from generating false alarms, as discussed in Section 5.1. To demonstrate this, we tested SFPD with the incomplete list of kernel objects identified by the basic traversal and observed more than 120 false alarms for a single memory snapshot.

6.3 GHOST

We evaluated GHOST's ability to identify objects hidden by rootkits, specifically processes and drivers. To detect hidden processes, we leveraged the Windows Task Manager as the internal source of information. We used the WinObj tool [26] to detect hidden drivers. The same experimental setup described in the SFPD experiments was used to collect memory snapshots from infected systems.

We tested GHOST with two real-world kernel-mode malware samples: FURootkit and Syzor.A. We ported the original Windows XP-based FURootkit to Windows Vista SP1.

GHOST correctly identified all hidden objects in both tests with *zero* false alarms. In the FURootkit test, GHOST easily identified the hidden process by checking the image file name at EPROCESS.ImageFileName and the process ID at EPROCESS.UniqueProcessId for each EPROCESS object. KOP identified the hidden process object by following a different pointer path than the one used by Windows Task Manager.

Syzor.A hides its own driver by zeroing out its driver object in memory. In the test with Syzor.A, GHOST identified the hidden driver object because it is empty and therefore it does not even have a driver name. So obviously it is not in the list returned by WinObj.

7. RELATED WORK

Kernel integrity has been the target of intense security research, given the increasing threat posed by kernel rootkits and other malware. Systems like CoPilot [20] and Livewire [8] passively scan the static portion of the kernel memory for integrity violations. More elaborate types of checks were also shown by Petroni *et al.* [21] to verify the semantic consistency of dynamic kernel structures based on manually created rules. State-based Control Flow Integrity [22] is similar to KOP as it also traverses the dynamic kernel object

graph. By using a simple type graph and manual annotations, it verifies the value of function pointers at each object it finds against some policy (e.g., pointing to a known module). Gibraltar [3] also relies purely on static type information and manual annotations to traverse the kernel memory for integrity checks. KOP represents a significant improvement over these approaches for its nearly complete coverage. Additionally, by leveraging KOP, SFPD is able to identify all malicious function pointers including the implicit ones in our experiments with real-world rootkits.

More preventive approaches towards kernel integrity include SecVisor [29], which enforces code integrity through the use of hypervisor-based memory protections. Lares [18] uses the same technique to guarantee the integrity of hooks deployed by anti-virus programs in the kernel code and data. NICKLE [24] proposes a hypervisor-based memory shadowing scheme to protect kernel code integrity and Patagonix [13] also relies on a hypervisor to trap code execution accesses and make sure that only legitimate code is executed.

Dynamic memory type inference has been addressed by work such as [23] with the goal of identifying heap corruption and type safety violations in C programs. Similar to KOP, their system also relies on type value constraints, type definitions, and an object graph traversal to map a program's heap. Others have approached the problem of inferring data structures from memory images without any access to source code or type definitions. Laika [5] uses Bayesian unsupervised learning to automatically infer the location and overall structure of the data objects used by user-level applications. Their approach is based on the observation that different types of data elements have values on different domain ranges. KOP also leverages this observation in resolving type ambiguities. Laika focuses on user-space programs. Operating systems have a much larger and more elaborate memory structure, for which their approach may not be suited. We believe that leveraging the source code is an important step to achieve high coverage and accuracy in analyzing kernel memory.

Pointer analysis for the C programming language has a long history [1, 2, 6, 9, 10, 19, 30, 31]. Its main goal has traditionally been performance. Thus, there has not been work that attempted to achieve both field-sensitivity and context-sensitivity with a codebase of millions lines of code. However, since our focus is precision rather than performance, we extended the algorithm described in [10] to achieve both field-sensitivity and context-sensitivity in KOP.

8. LIMITATIONS AND FUTURE WORK

Despite the encouraging results obtained from our evaluation, there is much room for improving KOP.

KOP's static analysis could be improved to automatically handle the kernel implementation corner cases discussed in Section 6 in a more general way. For example, tracking the arithmetic and logical operations associated with pointer values could provide a general way to identify bit manipulations in pointers. Likewise, identifying the use of casts in assignments could help us automatically determine implicit polymorphism. These improvements could make the task of porting KOP to a different OS easier.

The techniques used in KOP's memory analysis are also not perfect. Currently KOP relies on its knowledge of pointer fields to select a candidate from the range of possibilities. There are cases where this knowledge may not be sufficient to make the correct choice. It is very hard, for instance, to tell apart small objects with very few or no pointers at all, which may lead to inaccuracies in the traversal. One possibility to mitigate these problems is to increase the scope of our static analysis to determine domain constraints for other basic types in addition to pointers. For example, a unicode string should always be terminated by two consecutive null bytes, and enumerated (enum in C) types can only assume a statically-defined set of values. Such information would be very useful for increasing the precision when resolving type ambiguities.

One must also consider the possibility of an attacker trying to disrupt KOP's traversal by polluting the kernel memory. He could, for instance, intentionally break the internal structure of key kernel objects by tampering with the values stored at pointer fields. As a result, our traversal might incorrectly identify these objects due to pointer field mismatches. This attack is not as simple as it sounds, however, since the attacker has to carry it out in a way that the modifications do not destabilize the whole kernel and crash the system. Corrupting a pointer value which points to a string, for example, would likely be much less catastrophic than corrupting another one pointing to a scheduler queue, or another vital OS data structure. Our current system can tolerate this kind of attack up to a certain point, since it checks the pointer-value constraints in a flexible manner. However, it will not be able to do so if a very large number of pointers inside an object is manipulated. A more robust improvement could come from pre-determining which fields can be tampered with without crashing the system and ignoring them when matching pointer fields [7].

9. CONCLUSIONS

Dynamic kernel data have become a common target for malware looking to evade traditional code and static data-based integrity monitors. Previous solutions for inspecting dynamic kernel data can reach only a fraction of it, leaving holes which well-engineered malware can use for evasion. Thus, it is imperative that integrity protection systems be able to accurately and completely map kernel objects in memory.

In this paper we presented KOP, a system that can map dynamic kernel objects with very high coverage and accuracy by leveraging a set of novel techniques in static source code analysis and memory analysis. Our evaluation of KOP has shown substantial coverage gains over previous approaches. We implemented two integrity checking applications based on KOP to detect malicious function pointers and discover hidden objects. We evaluated them using real-world malware samples, demonstrating that KOP's high coverage and accuracy result in the ability to detect kernel integrity violations missed by previous approaches.

10. ACKNOWLEDGMENTS

We would like to thank Miguel Castro, Manuel Costa and Periklis Akritidis for kindly sharing their codebase with us; John Lin and Andy Ayers for their help with issues regarding Phoenix; Paul Royal and the Anti-Malware team in Microsoft for providing malware samples; Chris Hawblitzel, Ben Livshits and Bjarne Steensgaard for helpful discussions on implementing points-to analysis; David Evans and Helen Wang for their insightful comments on an early draft of this paper.

This material is based upon work supported in part by the National Science Foundation under Grant No. 0716570, Grant No. 0831300, Grant No. 0852131 and Grant No. 0855297, and the Department of Homeland Security under Contract No. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Homeland Security.

11. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Improving Software Security with a C Pointer Analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, May 2005.

[3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference*, 2008.

[4] Microsoft Corporation. Windows Research Kernel. `http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx`.

[5] 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*, 2008.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation (PLDI)*, 2000.

[7] 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)*, November 2009.

[8] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Tenth ISOC Symposium on Network and Distributed Systems Security (NDSS)*, February 2003.

[9] B. Hardekopf and C. Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Programming Language Design and Implementation (PLDI)*, 2007.

[10] N. Heintze and O. Tardieu. Ultra-Fast Aliasing Analysis using CLA - A Million Lines of C Code in a Second. In *Programming Language Design and Implementation (PLDI)*, 2001.

[11] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[12] S. Hultquist. Rootkits: The Next Big Enterprise Threat? `http://www.infoworld.com/article/07/04/30/18FErootkit_1.html`.

[13] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[14] Microsoft Corporation. Debugger engine and extensions api. `http://msdn.microsoft.com/en-us/library/cc267863.aspx`.

[15] Microsoft Corporation. Overview of Memory Dump File Options for Windows Server 2003, Windows XP, and Windows 2000. `http://support.microsoft.com/kb/254649`.

[16] Microsoft Corporation. Phoenix compiler framework. `http://connect.microsoft.com/Phoenix`.

[17] Offensive Computing. Public malware database. `http://www.offensivecomputing.net`.

[18] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[19] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient Field-Sensitive Pointer Analysis for C. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2004.

[20] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[21] N. L. Petroni Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[22] N. L. Petroni Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007.

[23] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic Heap Inference for Program Understanding and Debugging. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, 2007.

[24] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[25] Rootkit.com. `http://www.rootkit.com`.

[26] M. Russinovich. WinObj v2.15. `http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx`.

[27] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals (4th Edition)*. Microsoft Press, 2005.

[28] J. Rutkowska. klister. `http://www.rootkit.com/board_project_fused.php?did=proj14`.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[30] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

[31] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Programming Language Design and Implementation (PLDI)*, 1995.