CERIAS Tech Report 2010-04

GRAPH-BASED SIGNATURES FOR KERNEL DATA STRUCTURES

by Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, Xuxian Jiang

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Graph-based Signatures for Kernel Data Structures

Zhiqiang Lin[†], Junghwan Rhee[†], Xiangyu Zhang[†], Dongyan Xu[†], Xuxian Jiang[‡]
[†]Purdue University [‡]NCSU
{zlin, rhee, xyzhang, dxu}@cs.purdue.edu jiang@cs.ncsu.edu

ABSTRACT

The signature of a data structure reflects some unique properties of the data structure and therefore can be used to identify instances of the data structure in a memory image. Such signatures are important to many computer forensics applications. Existing approaches propose the use of value invariants of certain fields as data structure signatures. However, they do not fully exploit pointer fields as pointers are more dynamic in their value range. In this paper, we show that pointers and the topological properties induced by the points-to relations between data structures can be used as signatures. To demonstrate the idea, we develop SigGraph, a system that automatically extracts points-to relations from kernel data structure definitions and generates unique graph-based signatures for the data structures. These signatures are further refined by dynamic profiling to improve efficiency and robustness. Our experimental results show that the graph-based signatures achieve high accuracy in kernel data structure recognition, with zero false negative and close-to-zero false positives. We further show that SigGraph achieves strong robustness in the presence of malicious pointer manipulations.

1. INTRODUCTION

Given a kernel data structure definition, identifying instances of that data structure in a kernel memory image is a highly desirable capability in computer forensics, especially memory forensics where investigators try to recover semantic information from a memory image obtained from a suspect/victim machine [23, 10, 18, 32, 30]. Analogous to the pattern recognition problem in image processing, the problem of data structure – especially kernel data structure – instance recognition has received increasing attention. For example, the state-of-the-art solutions often rely on the *field value invariance* exhibited by a data structure as its signature [33, 31, 12, 8, 7]. The value of such a field is either constant or in a fixed range and the effectiveness and robustness of the value-invariant approach has been well demonstrated. However, there exist many kernel data structures that are not covered by the value-invariant approach. For example, some data structures do not have fields with invariant values or value ranges. It is also possible that an invariant-value field is corrupted (e.g., by kernel bugs or attacks), making the corresponding data structure instance un-recognizable. Furthermore, some value invariant-based signatures may not be unique enough to distinguish themselves from others. For example, a signature that demands the first field to have value 0 may generate a lot of false positives.

In this paper, we present a complementary model for kernel data structure signatures. Different from the value-invariant-based signatures, our approach, called SigGraph, uses a *graph structure rooted at a data structure* as the data structure's signature. More specifically, for a data structure with pointer field(s), each pointer field – identified by its offset from the start of the data structure – points to another data structure. Transitively, such points-to relations entail a graph structure rooted at the original data structure. We observe that data structures with pointer fields widely exist in operating system (OS) kernels. For example, when compiling the whole package of Linux kernel 2.6.18-1, we found that over 40% of all data structures have pointer field(s).

Compared with the field values of a data structure, the "topology" of such a "points-to" graph (which is across data structures) is much more stable. Moreover, in an OS kernel, it is unlikely that two different data structures have exactly the same graph-based signature, which is confirmed by our experiments with a number of Linux kernels. As such, SigGraph is deemed a natural scheme to uniquely identify kernel data structures with pointers.

SigGraph has the following key features: (1) It models the topological invariants between a subject data structure and those directly or transitively reachable via points-to relations. Furthermore, SigGraph recognizes and formulates the challenge that different data structures may share isomorphic structural patterns such that false positives are induced if the invariants are not properly chosen. SigGraph proposes a theoretically sound solution identifying signatures that are guaranteed not to cause false positives in ideal scenarios (e.g. pointers are always not null) and develops a number of practical extensions to adapt the algorithm to real-world scenarios (e.g. some pointers may be null). (2) SigGraph exploits the wealth of points-to relations between kernel data structures, and is able to generate *multiple* signatures for the same data structure. This is particularly powerful when operating under malicious pointer mutation attacks. (3) SigGraph avoids complex, expensive points-to analysis for `void` pointer handling (e.g., in [9]) as it can generate distinct signatures without involving those pointers. (4) The graph-based signatures can often be described by *context-free-grammars* such that parsers/scanners can be automatically generated to recognize data structure instances. More specifically, to determine if address x holds an instance of data structure T, we only need to perform pattern matching starting at x using the parser for T. This avoids the construction of (and dependence on) a *global* memory graph starting from the global variables and stack variables of a program/OS [17, 9].

We have performed extensive evaluation on SigGraph-generated signatures with several Linux kernels and verified the uniqueness of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

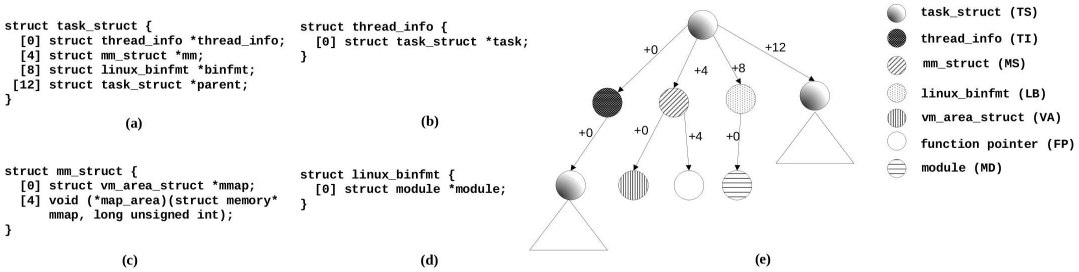


Figure 1: A working example of kernel data structures and a graph-based signature. The triangles represent recursions.

the signature. Our signatures achieve low false positives and zero false negatives when applied to data structure instance recognition on kernel memory images. Furthermore, our experiments show that SigGraph works in the absence of global memory maps and in the face of a range of kernel attacks that corrupt pointer fields. Finally, we conduct a security analysis to recognize SigGraph's strategic advantage and disadvantage when facing kernel attacks.

2. OVERVIEW

2.1 Problem Statement and Challenges

The goal of SigGraph is to use the inter-data structure topology induced by points-to relations as a data structure's signature. Consider 7 simplified Linux kernel data structures, 4 of which are shown in Figure 1(a)-(d). In particular, `task_struct(TS)` contains 4 pointers to `thread_info(TI)`,`mm_struct(MS)`,`linux_binfmt(LB)`, and `TS`, respectively. `TI` has a pointer to `TS`, and `MS` has two pointers: one points to `vm_area_struct(VA)` (not shown in the figure) and the other is a function pointer. `LB` has one pointer to `module(MD)`.

At runtime, if a pointer is not `null`, its target object should have the type of the pointer. Let $S_T(x)$ denote a boolean function that decides if the memory region starting at x is an instance of type T and let $*x$ denote the value stored at x. Take `task_struct` data structure as an example, we have the following rule, assuming all pointers are not `null`.

$$\begin{aligned} S_{\texttt{TS}}(x) \quad \rightarrow \quad & S_{\texttt{TI}}(*(x+0)) \ \wedge S_{\texttt{MS}}(*(x+4)) \ \wedge \\ & S_{\texttt{LB}}(*(x+8)) \ \wedge S_{\texttt{TS}}(*(x+12)) \end{aligned} \tag{1}$$

It means that if $S_{\texttt{TS}}(x)$ is true, then the four pointer fields must point to regions with the corresponding types and hence the boolean functions regarding these fields must be true. Similarly, we have the following

$$S_{\texttt{TI}}(x) \quad \rightarrow \quad S_{\texttt{TS}}(*(x+0)) \tag{2}$$

$$S_{\texttt{MS}}(x) \quad \rightarrow \quad S_{\texttt{VA}}(*(x+0)) \wedge S_{\texttt{FP}}(*(x+4)) \tag{3}$$

$$S_{\texttt{LB}}(x) \quad \rightarrow \quad S_{\texttt{MD}}(*(x+0)) \tag{4}$$

for `thread_info`, `mm_struct`, and `linux_binfmt`, respectively. Substituting symbols in rule (1) using rules (2), (3) and (4), we further have

$$\begin{aligned} S_{\texttt{TS}}(x) \quad \rightarrow \quad & S_{\texttt{TS}}(*(*(x+0)+0)) \ \wedge \ S_{\texttt{VA}}(*(*(x+4)+0)) \\ & S_{\texttt{FP}}(*(*(x+4)+4)) \ \wedge \ S_{\texttt{MD}}(*(*(x+8)+0))) \wedge \\ & S_{\texttt{TS}}(*(x+12)) \end{aligned} \tag{5}$$

The rule corresponds to the graph shown in Figure 1 (e), where the nodes represent pointer fields with their shapes denoting pointer types; the edges represent the points-to relations with their weights indicating the pointers' offsets; and the triangles represent recursive occurrences of the same pattern. It means that if the memory region starting at x is an instance of `task_struct`, the layout of the region must follow the graph's definition. Note that the inference of rule 5 is from left to right. However, we observe that the graph is so unique that the reverse inference tends to be true. In other words, we can use the graph as the signature of `task_struct` and achieve the *reverse* inference as follows.

$$\begin{aligned} S_{\texttt{TS}}(x) \quad \leftarrow \quad & S_{\texttt{TS}}(*(*(x+0)+0)) \ \wedge \ S_{\texttt{VA}}(*(*(x+4)+0)) \\ & \ldots \end{aligned} \tag{6}$$

To realize the SigGraph signature scheme we need to address a number of challenges:

- **Capturing uniqueness of signatures.** Given a static data structure definition, we aim to construct its points-to graph as shown in the `task_struct` example. However, it is possible that two distinct data structures may lead to *isomorphic* graphs which cannot be used to distinguish instances of the two data structures. Hence our first challenge is to identify the sufficient and necessary conditions for signature uniqueness between data structures.
- **Generating signatures.** It is possible that a data structure may have *multiple* unique signatures, depending on how (especially, how deep) the points-to edges are traversed when generating a signature. In particular, among the valid signatures of a data structure, finding the minimal signature that has the smallest size while retaining uniqueness (relative to other data structures) is a combinatorial optimization problem. Finally, it is desirable to *automatically* generate a parser for each signature that will perform the corresponding data structure instance recognition on a memory image.
- **Improving recognition accuracy.** Although statically a data structure may have a unique signature graph, at runtime, pointers may be `null` whereas non-pointer fields may have pointer-like values. As a result the data structure instances in a memory image may not fully match the signature. We need to handle such issues to improve recognition accuracy.

2.2 System Overview

The overview of the SigGraph system is shown in Figure 2. It consists of four key components: (1) data structure definition extractor, (2) dynamic profiler, (3) signature generator, and (4) parser generator. The starting point of our system is to extract data structure definitions of the target OS. In this paper, we use the source code of Linux kernels as subject programs. SigGraph extracts all kernel data structure definitions automatically through a compiler pass. Since our signatures rely on pointer fields whose values may be `null` or some special values, the dynamic *profiler* identifies such problematic pointer fields and handle them accordingly. The *signature generator* is responsible for checking if unique signatures exist for a data structure. If so, the generator will generate them. The generated signatures are passed to the *parser generator* component that can automatically generate parsers for individual data structures from their signatures.

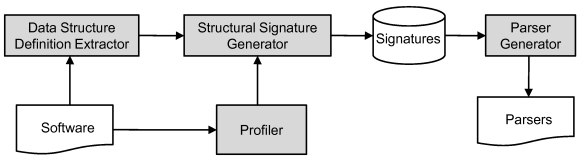


Figure 2: SigGraph system overview

```
struct A {
  [0]  struct B * a1;
    ...
  [12] struct C * a2;
    ...
  [18] struct D * a3;
}

struct X {
    ...
  [8]  struct Y  * x1;
    ...
  [36] struct BB * x2;
    ...
  [48] struct CC * x3;
    ...
  [54] struct DD * x4;
}

c80b20e0: 00 00 00 00 01 20 00 32  0a 00 00 00 00 ae ff 00
c80b20f0: c8 40 30 b0 00 00 00 00  00 10 00 00 c8 40 42 30
c80b2100: 00 00 c8 41 00 22 00 00  00 10 00 00 00 00 00 00
```

Figure 3: Insufficiency of pointer layout uniqueness.

3. DATA STRUCTURE DEFINITION EXTRACTION

A naive method to extract data structure definitions is to directly process source code. This method is error-prone because definitions may be present in multiple source files; they may have various scopes (i.e., the declaration contexts) and type aliases can be introduced by `typedef`. SigGraph's *data structure definition extractor* adopts a compiler-based approach, where the compiler is instrumented to walk through the source code and extract data structure definitions. It is robust as it is based on a full-fledged language frontend. In particular, we introduce a compiler pass in `gcc-4.2.4`. The pass takes abstract syntax trees (ASTs) as input as they retain substantial symbolic information [1]. The compiler-based approach also allows us to handle data structure in-lining, which occurs when a data structure has a field that is of the type of another structure; after compilation, the fields in the inner structure become fields in the outer structure. Furthermore, we can easily see through type aliases introduced by `typedef` via ASTs.

The net outcome of the compiler pass is the data structure definitions extracted in a canonical form. The pass is inserted into the compilation work flow right after data structure layout is finished (in `stor-layout.c`). During the pass, the AST of each data structure is traversed. If the data structure type is `struct` or `union`, its field type, offset, and size information is dumped to a file. To precisely reflect the field layout after in-lining, we flatten the nested definitions and adjust offsets.

We note that source code availability is *not* a fundamental requirement of SigGraph. For a close-source OS (e.g., Windows), if debug information is provided with the binary, SigGraph can simply use the debug information.

4. SIGNATURE GENERATION

Assume a data structure T has n pointer fields with offsets f_1, f_2, ..., f_n and types t_1, t_2, ..., t_n. A predicate $S_t(x)$ determines if the region starts at address x is an instance of t. The following production rule can be generated for T.

$$S_T(x) \rightarrow S_{t_1}(*(x+f_1)) \wedge S_{t_2}(*(x+f_2)) \wedge ... \wedge S_{t_n}(*(x+f_n)) \quad (7)$$

The problem of data structure signature generation is along the *reverse* direction of the above rule. Given a memory snapshot, we hope to identify instances of a data structure by trying to match the right-hand side of the rule (as a signature) with memory content starting at a certain location. Although it is generally difficult to infer the types of memory at individual locations based on the memory content, it is more feasible to infer if a memory location contains a pointer and hence to identify the layout of pointers with high confidence. This can be done recursively by following the pointers to the destination data structures. As such, the core challenge in signature generation is to find a finite graph induced by points-to relations (including pointers, pointer field offsets, and pointer types) that uniquely identifies a target data structure, which is the root of the graph. For convenience of discussion, we assume that pointers are not `null` and they each have an explicit type (i.e. not a `void *` pointer). We will describe how to handle those real-world issues in Section 6.

As mentioned earlier, two distinct data structures may have isomorphic structural patterns. For example, if two data structures have the same pointer field layout, we need to further look into the "next-hop" data structures (we call them *lower layer* data structures) via the points-to edges. Moreover, we observe that *even though the pointer field layout of a data structure may be unique (different from any other data structure), an instance of such layout in memory is not necessary an instance of the data structure*. Consider Figure 3, `struct A` and `X` have different layouts for their pointer fields. If the program has only these two data structures, it appears that we can use their one level pointer structure as the signature. However, this is not true. Consider the memory segment at the bottom of Figure 3, in which we detect three pointers (the boxed bytes). It appears that $S_A(0xc80b20f0)$ is true because it fits the one level structure of `struct A`. But it is possible that the three pointers are instead the instances of fields `x2`, `x3`, and `x4` in `struct X` and hence the region is part of an instance of `struct X`. In other words, a pattern scanner based on `struct A` will generate many false positives on `struct X`. The reason is that the structure of `A` coincides with the sub-structure of `X`. As we will show later in Section 7, such coincidences are very common.

To better model the issue, we introduce the concept of *immediate pointer pattern* (IPP) that describes the one level pointer structure as a string such that the aforementioned problem can be detected by deciding if an IPP is the substring of another IPP.

DEFINITION 1. *Given a data structure T, let its pointer field offsets be f_1, f_2, ..., and f_n, pointing to types t_1, t_2, ..., and t_n, resp. Its* immediate pointer pattern, *denoted as $IPP(T)$, is defined as follows.* $IPP(T) = f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot (f_3 - f_2) \cdot t_3 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n$.

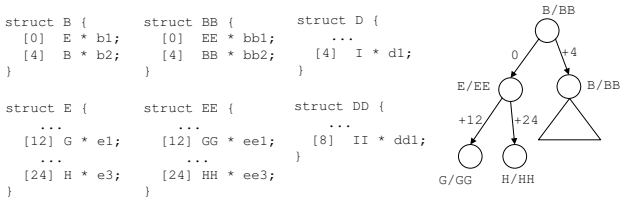
We say an $IPP(T)$ is a sub-pattern of $IPP(R)$ if $g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$ is a substring of $IPP(R)$, with $g_1 >= f_1$ and r_1, ..., r_n any pointer types.

Intuitively, an IPP describes the types of the pointer fields and their intervals. An $IPP(T)$ is a sub-pattern of $IPP(R)$ if the pattern of pointer field intervals of T is a sub-pattern of R's, disregard the types of the pointers. It also means that we cannot distinguish an instance of T from an instance of R in memory if we do not look into the lower layer structures. For instance in Figure 3, $IPP(A) = 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D$ and $IPP(X) = 8 \cdot Y \cdot 28 \cdot BB \cdot 12 \cdot CC \cdot 6 \cdot DD$. $IPP(A)$ is a sub-pattern of $IPP(X)$.

DEFINITION 2. *Replacing a type t in a pointer pattern with "$(IPP(t))$" is called one pointer expansion, denoted as $\xrightarrow{t}$.*

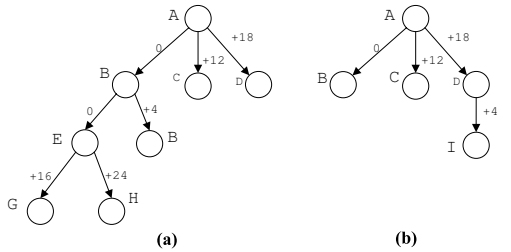
A pointer pattern of a data structure T is a string generated by a sequence of pointer expansions from $IPP(T)$.

For example, assume the definitions of B and D can be found in Figure 4.



(a) definitions **(b) structures of B and BB**

Figure 4: Data structure isomorphism.



(a) **(b)**

Figure 5: If the offset of field `e1` in `struct G` changes to 16, `struct A` has two possible signatures.

$$IPP(A) = \begin{array}{l} 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D \\ \xrightarrow{B} \boxed{0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot D}^{[1]} \\ \xrightarrow{D} \boxed{0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I)}^{[2]} \end{array} \quad (8)$$

The strings [1] and [2] are both pointer patterns of A. The pointer patterns of a data structure are candidates for its signature. As one data structure may have many pointer patterns, the challenge becomes to algorithmically identify the unique pointer patterns of a given data structure so that instances of the data structure can be identified from memory by looking for satisfactions of the pattern without causing false positives. If efficiency is a concern, the minimal pattern should also be identified.

Existence of Signature. The first question we need to answer is whether a unique pointer pattern exists for a given data structure. According to the previous discussion, given a data structure T, if its IPP is a sub-pattern of another data structure's IPP (including the case in which they are identical). We cannot use the one layer structure as the signature of T. We have to further use the lower-layer data structures to distinguish it from the other data structure. However, it is possible that T is not distinguishable from another data structure R if their structures are isomorphic.

DEFINITION 3. *Given two data structure T and R, let the pointer field offsets of T be f_1, f_2, ..., and f_n, pointing to types t_1, t_2, ..., and t_n, resp.; the pointer field offsets of R be g_1, g_2, ..., and g_m, pointing to types r_1, r_2, ..., and r_m, resp.*

T and R are isomorphic, denoted as $T \bowtie R$, if and only if

[1] $n \equiv m$;

[2] $\forall 1 \leq i \leq n \boxed{f_i \equiv g_i}^{[2.1]} \wedge (\boxed{t_i \bowtie r_i}^{[2.2]}$

$\vee \boxed{\textit{a cycle is formed when deciding } t_i \bowtie r_i}^{[2.3]})$;

Intuitively, two data structures are isomorphic, if they have the same number of pointer fields (Condition [1]) at the same offsets ([2.1]) and the types of the corresponding pointer fields are also isomorphic ([2.2]) or the recursive definition runs into cycles ([2.3]), e.g., when $t_i \equiv T \wedge r_i \equiv R$.

Figure 4 (a) presents the definitions of some data structures in Figure 3. The data structures whose definitions are missing from the two figures do not have pointer fields. According to Definition 3, $B \bowtie BB$ because they both have two pointers at the same offsets; and the types of the pointer fields are isomorphic either by the sub-structures ($E \bowtie EE$) or by the cycles ($B \bowtie BB$).

Given a data structure, we can now decide if it has a unique signature. As mentioned earlier, we assume that pointers are not `null` and are not of the `void*` type.

THEOREM 1. *Given a data structure T, if there does not exist a data structure R such that*

[1] *$IPP(T)$ is a sub-pattern of $IPP(R)$, and*

[2] *assume the sub-pattern in $IPP(R)$ is $g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$, $t_1 \bowtie r_1$, $t_2 \bowtie r_2$, ... and $t_n \bowtie r_n$,*

T must have a unique pointer pattern, that is, the pattern can not be generated from any other individual data structure through expansions.

The proof of Theorem 1 is presented in Appendix I. Intuitively, the theorem specifies that T must have a unique pointer pattern (i.e., a signature) as long as there is not an R such that $IPP(T)$ is a sub-pattern of $IPP(R)$ and the corresponding types are isomorphic.

If there is an R satisfying [1] and [2] in the theorem, no matter how many layers we inspect, the structure of T remains identical to part of the structure of R, which makes them indistinguishable. In Linux kernels, we have found a few hundred such cases (about 12% of overall data structures). But most of them are data structures that are rarely used or not important according to the kernel security and forensics literature.

Note that two isomorphic data structures may have different concrete pointer field types. But given a memory snapshot, it is unlikely for us to know the concrete types of memory cells. Hence, such information can not be used to distinguish the two data structures. In fact, concrete type information is not part of a pointer pattern. Their presence is rather for readability.

Consider the data structures in Figure 3 and Figure 4. Note all the data structures whose definitions are not shown do not have pointer fields. $IPP(A)$ is a sub-pattern of $IPP(X)$, $B \bowtie BB$ and $C \bowtie CC$. But D is not isomorphic to DD due to their different immediate pointer patterns. According to the theorem, there must be a unique signature for A. In this example, the pointer pattern [2] in Equation (8) is a unique signature and if we find pointers that have such structure in memory, they must indicate an instance of A.

Finding the Minimal Signature. Even though we can decide if a data structure T has a unique signature with the theorem, there may be multiple pointer patterns of T that can distinguish it from other data structures. Ideally, we want to find the minimal pattern as it requires the minimal parsing efforts during scanning. For example, if field `e1`'s offset in `struct G` is 16, `struct A` has two possible structures as shown in Figure 5. They correspond to the pointer patterns

$$0 \cdot (0 \cdot (16 \cdot G \cdot 8 \cdot H) \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot D$$

and

$$0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I)$$

One is generated by expanding B and then E, and the other is generated by expanding D. Either one can serve as a unique signature of A.

Algorithm 1 An approximate algorithm for signature generation

Input: Data structure T;
Output: The pointer pattern that serves as the signature.

1: s= $IPP(T)$
2: **let** $IPP(T)$ be $f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n$
3: **for** each sub-pattern p=$g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$ in $IPP(R)$ of a different structure R with $f_1 <= g_1$ **do**
4: $distinct$=$distinct \cup \{p\}$
5: **end for**
6: **while** $distinct \neq \phi$ **do**
7: s=expand(s)
8: **for** each $p \in distinct$ **do**
9: p=expand(p)
10: **if** p is different from s disregard type symbols **then**
11: $distinct$= $distinct - p$
12: **end if**
13: **end for**
14: **end while**
15: return s

expand(s)

1: **for** each type symbol $t \in s$ **do**
2: s= replace t with "$(IPP(t))$"
3: **end for**
4: return s

In general, finding the minimal unique signature is a combinatorial optimization problem: *given a data structure T, find the minimal pointer pattern of T that can not be a sub-pattern of any other data structure R, that is, cannot be generated by pointer expansions from a sub-pattern of $IPP(R)$*. The complexity of a general solution is likely in the NP category. In this paper, we propose an approximate algorithm that guarantees to find a unique signature if one exists, though the generated signature may not be the minimal one. It is a breadth-first algorithm that performs expansions for all pointer symbols on the same layer at one step until the pattern becomes unique.

The algorithm first identifies the set of data structures that may have $IPP(T)$ as their sub-patterns (lines 3-5). Such sub-patterns are stored in $distinct$. Next, it performs breadth-first expansions on the pointer pattern of T, stored in s, and the patterns in $distinct$, until all patterns can be distinguished. It is easy to infer that the algorithm will eventually find a unique pattern if one exists.

For the data structures in Figures 3 and 4, the pattern generated for A by the algorithm is

$$0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I) \tag{9}$$

It is produced by expanding B and D in $IPP(A)$.

Generating Multiple Signatures. In some scenarios, it is highly desirable to generate multiple signatures for the same data structure. A common scenario is that some pointer fields in a signature may not be dependable. For example, certain kernel malware may corrupt the values of some pointer fields and, as a result, the corresponding data structure instance will not be recognized by a signature that involves those pointers.

SigGraph mitigates such a problem by generating multiple unique signatures. In particular, if certain pointer fields in a data structure are potential targets of corruption, SigGraph will *avoid* using such fields during signature generation in Algorithm 1. For example, if field `e1`'s offset in `struct G` is 16 and field `a3` in `struct A` is not dependable. Algorithm 1 generates a signature for `struct A` by pruning the sub-graph rooted at field `a3` in Figure 5(a). The detailed algorithm is elided.

5. PARSER GENERATION

Given a data structure signature, i.e., a pointer pattern, our technique can automatically generate a parser. The parser will be used to scan memory images during forensic investigations and identify instances of the data structure. To automatically generate parsers, we describe all signatures using a context free grammar (CFG). Then we leverage `yacc` to generate parsers. The CFG is described as follows.

```
1 int isInstanceOf_A(void *x){
2   x=x+0;
3   {
4      y=*x;
5      y=y+0'
6      assertPointer(*y);
7      y=y+4;
8      assertPointer(*y);
9   }
10  x=x+12;
11   assertPointer(*x);
12   x=x+6;
13   {
14      y=*x;
15      y=y+4;
16      assertPointer(*y);
17   }
18   return 1;
19 }
```

Figure 6: The generated parser for `struct A`'s signature in Equation 9.

$$\begin{aligned} Signature &:= \textbf{number} \cdot Pointer \cdot Signature \mid \epsilon \\ Pointer &:= \textbf{type} \mid (Signature) \end{aligned} \tag{10}$$

In the above grammar, **number** and **type** are terminals that represent numbers and type symbols, respectively. A $Signature$ is a sequence of **number** $\cdot$ $Pointer$, in which $Pointer$ describes either the **type** or the $Signature$ of the data structure being pointed-to. It is easy to tell that the grammar describes all the pointer patterns in Section 4, including the signature of A generated by our technique (Equation (9)).

Parsers can be generated based on the grammar rules. Intuitively, when a **number** symbol is encountered, the field offset should be incremented by **number**. If a **type** is encountered, the parser asserts that the corresponding memory contain a pointer. If a '(' symbol is encountered, a pointer dereference is performed and the parser starts to parse the next level memory region until the matching ')' is encountered. A sample parser generated for the signature in Equation (9) can be found in Figure 6. Function `isInstanceOf_A` decides if a given address is an instance of A; `assertPointer` asserts the given address must contain a pointer value, otherwise an exception is thrown and the function `isInstranceOf_A` returns 0. The `yacc` rules to generate parsers are elided for brevity.

Considering non-pointer fields. So far a parser considers only the positive information from the signature, which is the fields that are supposed to be pointers, but does not consider the implicit negative information, which is the fields that are supposed to be non-pointers. In many cases, negative information is needed to construct robust parsers.

For example, assume a data structure T has a unique signature $0 \cdot A \cdot 8 \cdot B \cdot 4 \cdot C$. If there is a pointer array that stores a consecutive sequence of pointers, even though the signature is unique and has no structural conflict with any other data structures, the generated parser will mistakenly identify part of the array as an instance of T.

In order to handle such issues, the parser should also assert that the non-pointer fields must not contain pointers. Hence the parser for the above signature becomes the following. Method `assertNot Pointer` asserts that the given address does not contain a pointer.

```
1 int isInstanceOf_T(void *x){
2   x=x+0;
3   assertPointer(*x);    // field of type "A *"
4   x=x+4;
5   assertNotPointer(*x); // field of non-pointer
6   x=x+4;
7   assertPointer(*x);    // field of type "B *"
8   x=x+4;
9   assertPointer(*x);    // field of type "C *"
10 }
```

6. HANDLING PRACTICAL ISSUES

Kernel version	#Total structs	Signature Statistics					Number of Signatures in Different Steps												
		#Pointer structs	#Unique signature	Percent	Average depth	#No signature	1	2	3	4	5	6	7	8	9	10	11	12	13
2.6.15-1	8850	3597	3229	89.76%	2.31	368	1355	823	461	229	76	194	85	4	1	0	1	-	-
2.6.18-1	11800	4882	4305	88.18%	2.45	572	1820	1057	382	410	159	337	121	9	3	5	1	1	-
2.6.20-15	14992	6096	5395	88.50%	2.54	701	2137	1311	680	236	407	501	106	9	1	5	1	1	-
2.6.24-26	15901	6427	5645	87.83%	2.47	782	2172	1316	761	475	624	248	37	7	1	0	3	1	-
2.6.31-1	26799	9957	8683	87.20%	2.73	1274	3364	1951	696	319	1492	494	344	19	1	0	1	1	1

Table 1: Experimental results of signature uniqueness test

So far we have assumed an ideal environment for SigGraph. However, when applied to large system software such as the Linux kernel, SigGraph faces a number of practical challenges. In this section, we present our techniques to handle the following key issues.

1. **Null Pointers –** It is possible that pointer fields have a `null` value, which are not distinguishable from other non-pointer fields, such as integer or floating point fields with value 0. If 0s are considered as a pointer value, a memory region with all 0 values would satisfy any immediate pointer patterns, which is clearly undesirable.

2. **Void Pointers –** Some of the pointer fields may have a `void*` type and they will be resolved to different types at runtime. Obviously, our signature generation algorithm cannot handle such cases.

3. **User Level Pointers –** It is also possible that a kernel pointer field has a value that actually points to user space. For example, the `set_child_tid` and `clear_child_tid` fields in `task_struct`, and the `vdso` field in `mm_struct` point to user space. The difficulty is that that user space pointers have a very dynamic value range due to the very large user space, which makes it hard to distinguish them from non-pointer fields.

4. **Special Pointers –** A pointer field may have non-traditional pointer value. For example, for the widely used `list_head` data structure, Linux kernel uses `LIST_POISON1` with value `0x00100100` and `LIST_POSION2` with value `0x00200200` as two special pointer to verify that nobody uses non-initialized list entries. Another special value `SPINLOCK_MAGIC` with value `0xdead4ead` also widely spreads in some pointer fields such as in data structure `radix_tree`.

5. **Pointer Like Values –** Some of the non-pointer fields may have values that resemble pointers. For example, it is not a very uncommon coding style to cast a pointer to an integer field and later cast it back to a pointer.

6. **Undecided Pointers –** Union types allow multiple fields with different types to share the same memory location. This creates problems for us too when pointer fields are involved.

7. **Rarely Accessed Data Structures –** Our algorithm presented in Section 4 treats all data structures equally important and tries to find signatures that are unique regarding all data structures. However, some of the data structures are rarely used and hence the conflicts caused by them may not be so important.

We find that most of the above problems boil down to the difficulty in deciding if a field is pointer or non-pointer. Fortunately, the following observation leads to a simple solution: Pruning a few noisy pointer fields does not degenerate the uniqueness of the graph-based signatures. Even though a signature after pruning may conflict with some other data structure signatures, we can often perform a few more refinement steps to redeem the uniqueness. As such, we devise a dynamic profiling phase to eliminate the undependable pointer/non-pointer fields.

Our profiler relies on a virtual machine monitor QEMU [3] to keep track of kernel memory allocation and deallocation for kernel data structures. More specifically, since most kernel objects are managed by slab allocators, we hook the allocation and deallocation of `kmem_cache` objects through functions such as `kmem_cache_alloc` and `kmem_cache_zalloc`, retrieving function arguments and return values to track these objects. Their types are acquired by looking at their slab name tags. Then we track the life time of these objects, and monitor their values. More details on how to track the allocation/deallocation of kernel objects at the VMM level can be found in our technical report [28].

We monitor the values of a kernel data structure's fields to collect the following information: (1) How often a pointer field takes on a value different from a regular non-null pointer value; (2) How often a non-pointer field takes on a non-null pointer-like value; (3) How often a pointer has a value that points to the user space. In our experiment, we profile a number of kernel executions for long periods of time (hours to tens of hours).

With the above profiles, we revise our signature generation algorithm with the following refinements: (1) excluding all the data structures that have never been allocated in our profiling runs so that structural conflicts caused by these data structures can be ignored; (2) excluding all the pointer fields that have the `void*` type or fields of union types that involve pointers – in other words, these fields are considered undependable (Section 4), which is done by annotating with a special symbol. Note that they should *not* be considered as non-pointer fields either, so that method `assertNonPointer` discussed in Section 5 will not be applied to such fields; (3) excluding all the pointer fields that have ever had a `null` value or a non-pointer value during profiling; as well as all non-pointer fields that ever have a pointer value during profiling. Neither `assertPointer` nor `assertNonPointer` will be applied to these fields; (4) allowing pointers to have special value `0x00100100` or `0x00200200`.

7. EVALUATION

We have implemented a prototype of SigGraph, with C and Python code. Specifically, we instrument `gcc-4.2.4` to traverse ASTs and collect data structure definitions. Our parser generator is lex/yacc based, and the generated parsers are in C. The total implementation is around 9.5K lines of C code and 6.8K LOC python code.

7.1 Signature Uniqueness

We first test if unique signatures exist for kernel data structures. We take 5 popular Linux distributions (from Fedora Core 5 and 6; and Ubuntu 7.04, 8.04 and 9.10), and the corresponding kernel version are shown in the first column of Table 1. Then we compile these kernels using our instrumented `gcc`. Observe that there are quite a large number of data structures in different kernels, ranged from 8850 to 26799. Overall, we find nearly 40% of the data structures have pointer fields, and nearly 88% (shown in the 5^{th} column) of the data structures with pointer fields have unique signatures. We show the

Category	Static Properties of the Data Structure					SigGraph Signature				Value Invariant Signature			
	Data Structure Name	ID	Size	$\|F\|$	$\|P\|$	Statically-Derived		Dynamically-Refined		$\|Z\|$	$\|C\|$	$\|B\|$	$\|A\|$
						D	$\sum\|P\|$	D	$\sum\|P\|$				
Processes	task_struct	1	1408	354	81	1	81	2	233	269	17	55	244
	thread_info	2	56	15	4	2	91	2	45	5	2	4	5
	key	3	100	27	9	4	117	4	69	5	2	7	11
Memory	mm_struct	4	488	121	23	1	23	2	26	39	41	62	68
	vm_area_struct	5	84	21	10	4	1444	4	60	15	0	3	17
	shmem_inode_info	6	544	135	51	1	51	2	147	32	24	51	41
	kmem_cache	7	204	51	39	3	295	3	36	8	0	4	9
File System	files_struct	8	384	50	41	3	3810	3	13	38	4	8	9
	fs_struct	9	48	12	7	2	121	2	68	2	7	8	7
	file	10	164	40	11	5	17034	5	3699	15	4	12	17
	dentry	11	144	63	16	5	27270	5	1444	44	4	14	16
	proc_inode	12	452	112	49	1	49	3	455	27	16	33	41
	ext3_inode_info	13	612	151	58	1	58	2	166	59	27	50	53
	vfsmount	14	108	27	23	4	6690	4	1884	4	0	20	24
	inode_security_struct	15	60	16	6	7	277992	7	8426	1	1	3	2
	sysfs_dirent	16	44	11	7	4	1134	4	61	3	0	4	8
Network	socket_alloc	17	488	121	54	1	54	2	142	28	8	21	37
	socket	18	52	13	7	5	45907	5	2402	1	4	10	6
	sock	19	436	114	48	1	48	2	149	21	42	59	34
Others	bdev_inode	20	568	141	65	1	65	2	166	22	13	31	39
	mb_cache_entry	21	36	12	8	6	27848	6	6429	2	1	4	6
	signal_struct	22	412	99	25	2	395	2	90	41	30	38	44
	user_struct	23	52	13	4	6	586	6	394	1	0	1	2

Table 2: Summary of data structure signatures from Linux kernel 2.6.18-1

average steps of pointer pattern expansion needed to generate unique signatures (in the 6^{th} column). Because of graph isomorphism, there are data structures that do not have any unique signatures. The total number of such structures is shown in the 7^{th} column. Note that these are all static numbers before the dynamic refinement.

We also show the number of unique signatures at various steps of expansions from column 8 to column 21 in Table 1. For example, kernel 2.6.15-1 has 1355 data structures that have unique one-level signatures and 823 data structures that have unique two-level signatures.

7.2 Effectiveness Compared with Value-invariant

To test the effectiveness of SigGraph, we take Linux kernel 2.6.18-1 as a working system, and show how the generated signatures can detect data structure instances. We list 23 widely used kernel data structures which are shown in the 2^{nd} column of Table 2. We choose these data structures because: (1) they are the most commonly examined data structures in existing literature [23, 10, 18, 32, 30, 33, 31, 8]; (2) they are very important data structures that can show the status of the system (*the primary focus of memory forensics*) in the aspects of process, memory, network and file system; from these data structures, we can reach most of the kernel objects; and (3) they contain pointer fields. Note that when parsing instances for these data structures, other data structures are also traversed, as our signatures often contain lower level data structures.

To ease our presentation, we assign an ID to each data structure, which is shown in the 3^{rd} column of Table 2, we use F to represent the set of fine-grained fields, and P to represent the set of pointer fields. A fine-grained field is a field with a primitive type (not a composite data type such as a `struct` or an array). Then, we present the corresponding total number of fields $|F|$ and pointers $|P|$ in the 5^{th} and 6^{th} columns, respectively.

7.2.1 Experiment Setup

We performed two sets of experiments. We first use our profiler to automatically prune the noisy pointer/non-pointer fields, generate refined signatures, and then detect the instances. After that we perform a comparison with value invariant based signatures to further confirm the effectiveness of our system.

Memory Snapshot Collection The first input to the effectiveness test is the snapshots of physical memory, which are acquired by instrumenting QEMU [3] to dump them on demand. We set the size of the physical RAM to 256M.

Ground Truth Acquisition The second input is the ground truth data of the kernel objects under test. We leverage and modify a kernel dump analysis tool, the RedHat `crash` utility [2], to analyze our physical memory image and collect the ground truth, through a data structure instance query interface driven by a python script. Note that to enable the `crash` dump analysis, the kernel needs to be rebuilt with debug information.

7.2.2 Dynamic Refinement

In this experiment, we carry out the dynamic refinement phase as described in Section 6. The depth and the size of signatures before and after pruning are presented in the "SigGraph Signature" columns in Table 2, with D the depth and $\sum|P|$ the number of pointer fields. Note that the signature generation algorithm has to be run again on the pruned data structure definitions to ensure uniqueness. Observe that since pointer fields are pruned and hence the graph topology gets changed, our algorithm has to perform a few more expansions to redeem uniqueness, and hence the depth of signatures increases after pruning for some data structures, such as `task_struct`.

7.2.3 Value Invariant based Signatures

To compare our approach with value invariant based signatures [33, 31, 12, 8], we also implemented a basic value-invariant signature generation system. In particular, we generally derive four types of invariants for each field, (1) zero-subset: a field is included if it is always zero across all instances during training runs; (2) constant: a field is always constant; (3) bitwise-AND: the bitwise AND of all values of a field is not zero, that is, they have some non-zero common bits; and (4) alignment: if all instances of a field are well-aligned at a power of two (other than 1) number.

To derive these value invariants, we perform two types of profiling: one is access frequency profiling (to prune out the fields that are

never accessed by the kernel), the other is to sample their values and produce the signatures. The access frequency profiling is achieved by instrumenting QEMU to track memory reads and writes. Sampling is similar to the sampling in our dynamic refinement phase.

All the data structures have value invariants, and the statistics of these signatures are provided in the last four column of Table 2. The total numbers of zero-subset, constant, bitwise-AND, and alignment are denoted as $|Z|$, $|C|$, $|B|$, and $|A|$, respectively.

7.2.4 Results

The final results for each signature when scanning a test image is shown in Table 3. The second column shows the total number of true instances of the data structure, which is acquired by the modified `crash` utility [2]. The $|R|$ column shows the number of instances the signature scanning detected. Due to the limitation of `crash`, these objects have to be live, i.e., reachable from global or stack variables. However, signature approaches are able to identify free objects. Hence, the detected free objects are determined as false positives (FPs) based on the ground truth from `crash`. We further take the free but not-yet-overwritten objects, which can also be traversed by `crash` if the slab allocator haven't released it to free pages, to better evaluate the real FPs. We present the FPs without considering free objects in the $|FP|$ column and the FPs considering free objects in the $|FP'|$ column. The false negative FN is computed by comparing with the ground truth objects from `crash`.

Observe that among the examined 23 data structures, when free objects are not considered, there are 16 that our approach precisely and completely identifies all instances (both FP and FN are zero). However, value invariant has only 5 such data structures. If we consider free objects, 20 can be perfectly identified with our approach, whereas 9 can be detected via the value based approach. Note for value invariant signature systems, high false positive rates imply the derived signatures are hardly useable. In other words, value invariant systems may report no signature for these data structures. For the 23 data structure we listed, SigGraph produces high quality signatures.

From the table, we could see our system has false positive for three of the data structures, in particular the `vm_area_struct` (ID 5 in Table 3), `dentry` (ID 11) and `sysfs_dirent` (ID 16). We carefully examined the corresponding snapshot, and found the reason is that some of them are truly freed object (the `dentry` case), and some of them are caused by our profiling that failed to capture some ground truth data (for `vm_area_struct` and `sysfs_dirent`). The detailed false positive analysis on all these cases is presented in Appendix II.

Summary No FNs are observed for our approach, while some are observed for the value invariant based approach. Our approach also has a very low FP rate. We believe the reasons are the following. (1) Graph based signatures are more informative as they include information of data structures at lower levels whereas value-based signatures only look at one level (namely the fields of the data structure itself). (2) Graph-base signatures are more stable and their uniqueness can be algorithmically determined, that is, we can expand the signature along points-to edges as many times as we want to achieve uniqueness, which is hard to perform for value-based signatures.

More Capability in Memory Forensics We also observe that signature approaches, including both our approach and value-invariant based approaches, can be used to identify free objects, such as in the cases of `mm_struct` and `fs_struct` data structures. However, our approach can perform better and have identified more free objects, such as in the case of `inode_security`, `dentry`, and `mb_cache_entry`, than previous approaches. Note that object traversal based approach is not able to identify free objects.

ID	$\|I\|$	SigGraph Signature				Value-Invariant			
		$\|R\|$	FP	FP'	FN	$\|R\|$	FP	FP'	FN
1	88	88	0.00	0.00	0.00	88	0.00	0.00	0.00
2	88	88	0.00	0.00	0.00	93	6.45	6.45	1.08
3	22	22	0.00	0.00	0.00	19	0.00	0.00	15.79
4	52	54	3.70	0.00	0.00	55	5.45	0.00	0.00
5	2174	2233	2.64	0.40	0.00	2405	9.61	7.52	0.00
6	232	232	0.00	0.00	0.00	226	0.00	0.00	2.65
7	127	127	0.00	0.00	0.00	5124	97.52	97.52	0.00
8	53	53	0.00	0.00	0.00	50	0.00	0.00	6.00
9	52	60	13.33	0.00	0.00	60	13.33	0.00	0.00
10	791	791	0.00	0.00	0.00	791	0.00	0.00	0.00
11	31816	38611	17.60	0.01	0.00	31816	0.00	0.00	0.00
12	885	885	0.00	0.00	0.00	470	0.00	0.00	88.30
13	38153	38153	0.00	0.00	0.00	38153	0.00	0.00	0.00
14	28	28	0.00	0.00	0.00	28	0.00	0.00	0.00
15	40067	40365	0.74	0.00	0.00	142290	71.84	70.93	0.00
16	2105	2116	0.52	0.52	0.00	88823	97.63	97.63	0.00
17	75	75	0.00	0.00	0.00	75	0.00	0.00	0.00
18	55	55	0.00	0.00	0.00	49	0.00	0.00	12.24
19	55	55	0.00	0.00	0.00	43	0.00	0.00	27.90
20	25	25	0.00	0.00	0.00	24	0.00	0.00	4.17
21	520	633	17.85	0.00	0.00	638	18.50	0.00	0.00
22	73	73	0.00	0.00	0.00	72	0.00	0.00	1.39
23	10	10	0.00	0.00	0.00	10591	99.91	99.91	0.00

Table 3: Experimental results of our graph based signature and value-invariant signature

7.3 Effectiveness without Memory Graph

Memory graph based techniques construct a global reference graph that connects all live objects. The roots are global and stack variables. Objects become invisible if pointers are corrupted so that they are not reachable from the roots. In contrast, SigGraph explores individual graph patterns related to the provided data structure, without constructing a global graph. Therefore, we believe SigGraph provides more robustness against global/stack variable corruption.

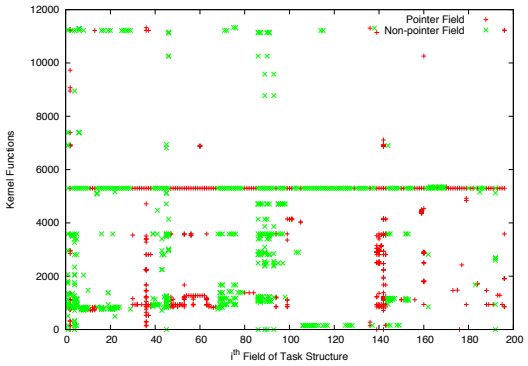
To verify our claim, we developed a rootkit to test this feature. In particular, the rootkit (a kernel module per se) overwrites several global variables which are related to process list and the management of slab cache, including `pid_hash`, `init_task`, and `task_struct_cachep`.

Before loading our rootkit, we took a snapshot, which has 78 running process. Then we run our rootkit. The system crashed as expected due to the pointer corruption. We took another snapshot. Next, we run the `crash` utility (which takes a memory graph-based approach) on these two images, for the first image, `crash` reported there are 78 processes, but for the second one, it reported "invalid kernel virtual address: 0 type: `pid_hash` content".

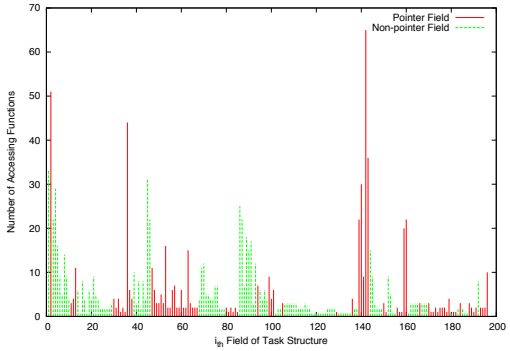
Then, we run our `task_struct` signature parser to scan the process instances, from the first image, we identified 78, and from the second image, we reported 77 instances – this is because `init_task` (which is an instance of `task_struct`) has been cleared.

7.4 Multiple Signatures

One powerful feature of SigGraph is that multiple signatures can be generated for the same data structure. We perform the following experiment with `task_struct` data structure. In each run of the experiment, we exclude one of the 38 pointer fields of `task_struct` (assuming that the pointer is corrupted) before running Algorithm 1. In each of the 38 runs, the algorithm is still able to compute a unique, alternative signature for `task_struct`. Next, we increase the number of corrupted pointer fields from 1 to 2, and conduct C_{38}^2 runs of Algorithm 1 (exhausting the combinations of the two pointers excluded). The algorithm is still able to generate a valid signature for each run.



(a) Detailed field access functions



(b) Statistics on field access functions

Figure 7: Profiling result on task_struct field access

The above experiments indicate that SigGraph is robust in the face of corrupted (excluded) pointer fields. However, the robustness has its limit. As the other extreme, we exclude 37 of the 38 pointer fields of `task_struct` and conduct $C_{38}^{37} = 38$ runs of Algorithm 1. Among the 38 runs, Algorithm 1 only generates valid signatures in 4 runs, where one of the following pointers is retained: `fs_struct`, `files_struct`, `namespace`, and `signal_struct`.

7.5 Performance Overhead

SigGraph is mainly used in off-line memory forensic analysis and thus its performance overhead is not so critical. Still, we measure the performance overhead of our scanners generated by SigGraph. The detailed results are shown in Appendix III. We believe that the performance overhead is still reasonable. In particular, when the signatures' depth is small, our scanners incur 10X to 20X overhead than value-invariant scanners.

8. SECURITY ANALYSIS

In this section, we analyze how SigGraph can be used in a more hostile environment, namely, in the presence of kernel level malware.

First, we study how SigGraph performs when the kernel is under attack by kernel malware. We studied the 23 kernel rootkits in [29]. We found that the majority of these rootkits (e.g., `adore-ng-0.56` and `enyelkm-1.2`) involve kernel hook (function pointer) hijacking in order to achieve their goal of hiding objects (e.g., processes, files, and network connections). Since they do not directly manipulate the kernel data structure instances, SigGraph signatures/parsers will be able to recover relevant kernel data structure instances from the memory image of a kernel under attack.

However, there exist rootkits that directly manipulate kernel data structure values – especially those of the pointer fields. For example, `fuuld`, `linux-fu` and `hp-1.0.0` manipulate the `next` and the `previous` pointers in `task_struct` at offsets 128 and 132 so that the `task_struct` for the malicious process is disconnected from the `tasks` list. Yet SigGraph is still able to recover *all* instances of `task_struct` (including the disconnected ones) from the memory image, thanks to the existence of multiple alternative signatures of `task_struct` in the absence of those corrupted pointers (Section 7.4). We have performed experiments to confirm the success of SigGraph in the face of these rootkits (demonstrated in the video clip submitted). In particular, the alternative `task_struct` signature has $\sum |P| = 221$.

While SigGraph has no problem handling the existing rootkits, we envision that there may be more sophisticated attempts to evade SigGraph in the future. We will discuss them in the following. We assume that the attacker has knowledge about SigGraph and has gained control of the kernel. Evasion can be approached by manipulating *pointer fields* or *non-pointer fields*.

Malicious Pointer Value Manipulation Since SigGraph relies on inter-data structure topology induced by pointers, manipulating pointers would be a natural attempt to defeat SigGraph. However, compared to non-pointer values, pointers are more sensitive to mutation because any changes to a pointer value may very likely lead to kernel crashes. Note that re-pointing a pointer to another data structure instance of the same type may not affect SigGraph in discovering the mutated instance. While the attacker may try to manipulate pointer fields that are not used, recall that SigGraph has a dynamic refinement phase that gets rid of such unused or undependable fields before signature generation.

The attack may try extra hard by destroying a pointer field after a reference, and then restoring it before its next reference. As such, it is likely for a snapshot not to see the true value, depending on the timing. However, carrying out such attacks is challenging as there may be many places that access the pointer field. All such places need to be patched in order to respect the original semantics of the kernel. We can anticipate that demands a complex and expensive static analysis on the kernel. To achieve an under-approximation of the demanded efforts. We conducted a profiling experiment on `task_struct`. We collect the functions that access each field, including both pointers and non-pointers. The results are shown in Figure 7(a), We observe that on average fields are accessed by at least 6 functions. Some fields have even been accessed by 70 functions (the statistics is shown in Figure 7(b)). Note that these are only profiled numbers, the static counterparts may be even larger. Even if the attacker achieves some success, SigGraph can still leverage its multiple signature capability to avoid using pointers that are easily manipulatable.

Malicious Non-Pointer Value Manipulation Another possible way to confuse SigGraph is to mutate a non-pointer value to resemble that of a pointer. SigGraph has also built-in protection against such attacks. First of all, the dynamic refinement phase will get rid of most fields that are vulnerable to such mutation. Moreover, compared to mutation within a domain, such as changing an integer field (with the range from 1 to 100) from 55 to 56, cross-domain mutation, such as changing the integer field to a pointer, has a much higher chance to crash the system. Hence, we suspect that not many non-pointer fields are susceptible. In the future, we plan to use fuzzing, similar to [12], to study how many fields allow such cross domain value mutation. In fact, we can simply *integrate* SigGraph signatures with the value-invariant signatures (e.g., those derived by [12]) for the same data structure, which is likely to achieve stronger robustness against malicious non-pointer manipulation.

Other Possible Attacks The attacker can change data structure layout

to evade SigGraph. However, such attacks are challenging. He/she needs to intercept the corresponding kernel object allocations and deallocations to change layout at runtime. Furthermore, all accesses to the affected fields need to be patched. Without knowing how the layout is changed, SigGraph will fail.

Furthermore, the attacker could also try to generate *fake* data structure instances to thwart the use of SigGraph. However, we should point out that fake instance creation is a generally hard problem across all signature-based approaches, including the value-invariant approaches. In fact, SigGraph makes such attacks harder as the attacker would have to fake the *multiple* data structures involved in a graph signature and make sure that all the points-to relations among these data structures are setup properly.

9. RELATED WORK

Memory Forensics Memory forensics is a process of analyzing a memory image to explain the current state of a computer system. It has been evolving from basic techniques such as string search to more complex methods such as object traversal (e.g.,[23, 30, 10, 18, 9]) and signature based scanning (e.g., [33, 31, 12, 8, 4]).

Object traversal techniques search memory by walking through OS data structures. They rely on building a whole reference graph of all data structures. Hence, they mostly work for live data because "dead" (i.e. freed) data cannot be reached by the reference graph. Constructing reference graphs relies on precisely resolving types of memory objects, which is often hard in the presence of `void` pointers or unknown data structures. For example, kernel objects that are part of a rootkit data structure cannot be traversed via the reference graph as the rootkit data structure type is unknown, even though the kernel data structure definitions themselves are known. Also, if a pointer in the reference graph is corrupted, then the memory region being pointed to cannot be visited. However, in SigGraph, we can avoid such problems as we do not rely on a fully connected global reference graph.

Signature scanning directly searches memory using signatures. In particular, Schuster [31] presented PTfinder for linearly searching Windows memory to discover process and thread structures, using manually created signatures. Similar to PTfinder, GREPEXEC [4], Volatility [33], Memparser [8] are the other systems that have more capabilities of searching other objects. As signatures are the key to these system, Dolan-Gavitt at el. [12] proposed an automated way to derive robust data structure signatures. SigGraph complements these systems by deriving another scheme for data structure signature generation.

Rootkit Detection Kernel-level rootkits pose a significant threat to the integrity of operating systems. Earlier research uses specification based approach deployed in hardware (e.g., [34, 21]), virtual machine monitor (e.g., Livewire [14]), or binary analysis [16] to detect kernel integrity violations. Recent advance includes state-based control flow integrity checking (e.g., SBCFI [24] and KOP [9]), and data structure invariant based checking (e.g., [22, 7, 12]).

Our work is inspired by the data structure invariant detection, and hence closely related to [22, 7, 12]. In particular, Petroni et al. [22] proposed examining semantic invariants (such as a process must be on either the wait queue or the run queue) of kernel data structures to detect rootkits. The key observation is that any violations of semantic invariants indicate rootkit presence. However, the extraction of semantic invariants was based on manually created rules. Afterwards, Baliga et al. [7] presented using dynamic invariant detector Daikon [13] to extract data structure constraints. The invariants detected include membership, non-zero, bounds, length, and subset relations. In contrast, we focus on structural patterns. We believe our approach is complementary to theirs. Most recently, Dolan-Gavitt at el. [12] proposed a novel system to automatically select robust signatures for kernel object signatures. Their observation is that value-invariants could be evaded by attackers and thus they propose to use fuzzing technique to test the robustness of value-invariants. The key difference is that they focus on value invariants while we focus on pointer-induced topological patterns between data structures. As discussed earlier, the best practice is likely the *integration* of the two approaches.

Malware Signature Derivation based on Data Structure Pattern Data structures are one of the important and intrinsic properties of a program. Recent advance has demonstrated that data structure patterns can be used as program signature. In particular, Laika [11] shows a way of inferring the layout of data structure from snapshot, and use the layout as signature. Their inference is based on an unsupervised Bayesian learning and they assume no prior knowledge about program data structures. Laika and SigGraph are substantially different: (1) Laika focuses on how to derive a program signature from data structure patterns, whereas SigGraph focuses on how to discover data structure instances from data structure patterns and how to derive such patterns. (2) Technically, Laika does not aim to accurately infer the data structure patterns (a large number of wrong layout could not hurt their system as they just want to have a classifier to classify the program), however we have to accurately match the instance, otherwise the high false positive and the high false negative ratios would render the system unusable. (3) Laika is not interested in any particular type of objects, whereas the output of SigGraph is the specific data structure instances.

Data Structure Type Inference There is a large body of research in program data structure type inference, such as object oriented type inference [20], and aggregate structure identification [26], binary static analysis based type inference [5, 6, 27], abstract type inference [19, 15], and dynamic heap type inference [25]. Most these techniques are static, aiming to infer types of unknown objects in code. SigGraph is more relevant to dynamic techniques. Dynamic heap type inference by Polishchuk et al. [25] focuses on typing heap objects in memory, using various constraints, such as size and type constraints. Heap object types are decided by resolving these constraints. The difference between their work and ours is that they assume all the heap objects are known, including their sizes and locations in the heap. This is done through instrumentation. However, SigGraph's input is simply a memory image.

10. CONCLUSION

In this paper, we have demonstrated that the points-to graphs between data structures can be used as data structure signatures. We present SigGraph, a system that automatically derives such graph-based data structure signatures. It complements the value-invariant signature systems by covering the majority of kernel data structures with pointer fields. Our experiments show that the signatures generated by SigGraph achieve zero false negative rate and very low false positive rate. Moreover, the signatures are not affected by the absence of global memory graphs and are robust against the corruption of pointer fields. For best effect, we advocate the combination of our graph-based signatures and the value-invariant-based signatures.

11. REFERENCES

[1] Gnu compiler collection (gcc) internals. *http://gcc.gnu.org/onlinedocs/gccint/*.

[2] Mission critical linux. In Memory Core Dump, http://oss.missioncriticallinux.com/projects/mcore/.

[3] QEMU: an open source processor emulator. *http://www.qemu.org/*.

[4] bugcheck. grepexec: Grepping executive objects from pool memory. *Uninformed Journal 4* (2006).

[5] BALAKRISHNAN, G., , BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'04)* (2004), Springer-Verlag, pp. 5–23.

[6] BALAKRISHNAN, G., AND REPS, T. Divine: Discovering variables in executables. In *Proceedings of International Conf. on Verification Model Checking and Abstract Interpretation (VMCAI'07)* (Nice, France, 2007), ACM Press.

[7] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)* (Anaheim, California, December 2008), pp. 77–86.

[8] BETZ, C. Memparser. http://sourceforge.net/projects/memparser.

[9] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)* (Chicago, IL, USA, 2009), pp. 555–565.

[10] CASE, A., CRISTINA, A., MARZIALE, L., RICHARD, G. G., AND ROUSSEV, V. Face: Automated digital evidence discovery and correlation. *Digital Investigation 5*, Supplement 1 (2008), S65 – S75. The Proceedings of the Eighth Annual DFRWS Conference.

[11] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)* (San Diego, CA, December, 2008), pp. 231–244.

[12] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)* (Chicago, Illinois, USA, 2009), ACM, pp. 566–577.

[13] ERNST, M., COCKRELL, J., GRISWOLD, W., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering 27*, 2 (2001), 1–25.

[14] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium (NDSS'03)* (February 2003).

[15] GUO, P. J., PERKINS, J. H., MCCAMANT, S., AND ERNST, M. D. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA'06)* (Portland, Maine, USA, 2006), ACM, pp. 255–265.

[16] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference(ACSAC'04)* (2004), pp. 91–100.

[17] 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).

[18] MOVALL, P., NELSON, W., AND WETZSTEIN, S. Linux physical memory analysis. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference* (Anaheim, CA, 2005), USENIX Association, pp. 23–32.

[19] O'CALLAHAN, R., AND JACKSON, D. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th international conference on Software engineering* (Boston, Massachusetts, United States, 1997), ACM, pp. 338–348.

[20] PALSBERG, J., AND SCHWARTZBACH, M. I. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications* (Phoenix, Arizona, United States, 1991), ACM, pp. 146–161.

[21] PETRONI, N. L., JR., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, August 2004), pp. 179–194.

[22] PETRONI, N. L., JR., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium* (Vancouver, B.C., Canada, August 2006), USENIX Association.

[23] PETRONI, N. L., JR., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3*, 4 (2006), 197 – 210.

[24] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)* (Alexandria, Virginia, USA, October 2007), ACM, pp. 103–115.

[25] POLISHCHUK, M., LIBLIT, B., AND SCHULZE, C. W. Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not. 42*, 1 (2007), 39–46.

[26] RAMALINGAM, G., FIELD, J., AND TIP, F. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)* (San Antonio, Texas, 1999), ACM, pp. 119–132.

[27] REPS, T. W., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'08)* (2008), pp. 16–35.

[28] RHEE, J., AND XU, D. Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. *Technical Report CERIAS 2010-02, Purdue University* (2010).

[29] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection, Cambridge, MA, USA* (2008), pp. 1–20.

[30] RUTKOWSKA, J. Klister v0.3. https://www.rootkit.com/newsread.php?newsid=51.

[31] SCHUSTER, A. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation 3*, Supplement-1 (2006), 10–16.

[32] SUTHERLAND, I., EVANS, J., TRYFONAS, T., AND BLYTH, A. Acquiring volatile operating system data tools and techniques. *SIGOPS Operating System Review 42*, 3 (2008), 65–73.

[33] WALTERS, A. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[34] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th ACM SIGOPS European workshop* (Saint-Emilion, France, 2002), ACM, pp. 239–242.

```
struct task_struct{                    struct vm_area_struct {
   [156] struct mm_struct *active_mm;     [0] struct mm_struct *vm_mm;
   [160] struct linux_binfmt *binfmt;     [4] long unsigned int vm_start;
   [164] long int exit_state;             [8] long unsigned int vm_end;
   [168] int exit_code;                  [12] struct vm_area_struct *vm_next;
   [172] int exit_signal;                [16] pgprot_t vm_page_prot;
   [176] int pdeath_signal;              [20] long unsigned int vm_flags;
   [180] long unsigned int personalit; ...
}                                      }

0xc035dc9c <init_task+156>: 0xce8e04e0 0x00000000 0x00000000 0x00000000
0xc035dcac <init_task+172>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dcbc <init_task+188>: 0x00000000 0x00000000 0xc035dc00 0xc035dc00
0xc035dccc <init_task+204>: 0xc12f1704 0xc12f1704 0xc035dcd4 0xc035dcd4
0xc035dcdc <init_task+220>: 0xc035dc00 0x00000000 0x00000000 0x00000000
0xc035dcec <init_task+236>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dcfc <init_task+252>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dd0c <init_task+268>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dd1c <init_task+284>: 0x00000000 0x02bf54e4 0x00000000 0x002eff84
0xc035dd2c <init_task+300>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dd3c <init_task+316>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc035dd4c <init_task+332>: 0xc035dd4c 0xc035dd4c 0xc035dd54 0xc035dd54
```

Figure 8: False Positive Analysis of `vm_area_struct`.

Appendix I: The Proof of Theorem 1

PROOF. For each data structure R different from T, either condition [1] or [2] is not satisfied according to the preconditions of the theorem.

If [1] is not satisfied, $IPP(T)$ can be used to distinguish T from R.

If [2] is not satisfied, there must be an i such that t_i is not isomorphic to r_i. There must be a minimal k, after k level of expansions, the pointer pattern of t_i is different from r_i's, disregard the type symbols. We say one level of expansion is to expand along all type symbols for one step. $IPP(T)$ can be considered as the pointer pattern of T with $k = 0$ level of expansion.

Since there are finite number of data structures, we can always identify the maximal among all the k values. Lets denote it as k_{max}. Hence, the pointer pattern of T after k_{max} levels of expansions can distinguish T from any other individual data structure. □

Appendix II: False Positive Analysis

In this appendix, we analyze the three false positive cases in detail for data structure `vm_area_struct`, `dentry` and `sysfs_dirent`.

For `vm_area_struct`, we have 9 false positives among the total 2233 detected instances. After dynamic refinement, some pointer fields are pruned, such as the pointer field at offset 12 (as shown in Figure 8). Finally, the generated unique signature contains only the first layer pointer structure, in particular, it consists of a pointer field at offset 0 (`mm_struct`), and then a sequence of non-pointer fields, and so on. However, the `task_structure` starting from offset 156 has the identical sub-pattern except the offset 160 is a pointer. But in some rare occasions (which are not captured by our profiler), the pointer field at offset 160 could be 0, leading to a false positive. This is due to the difference between the training images and the test image. We find 9 FPs in this case.

We have 2 FPs for `dentry`, which are shown in Figure 9(a). We classify these two instances as FPs because they cannot be found in either the pool of live objects or the pool of free objects. However, if

```
struct dentry {
    [0] atomic_t d_count;
    [4] unsigned int d_flags;
    [8] raw_spinlock_t raw_lock;
   [12] unsigned int magic;
   [16] unsigned int owner_cpu;
   [20] void *owner;
   [24] struct inode *d_inode;
   [28] struct hlist_node d_hash;
   [36] struct dentry *d_parent;
   ...
   [84] long unsigned int d_time;
   [88] struct dentry_operations *d_op;
   ...
}

fp1
0xc72bdf48: 0x00000000  0x00000010  0x00000001  0xdead4ead
0xc72bdf58: 0xffffffff  0xffffffff  0x00000000  0x00000000
0xc72bdf68: 0x00200200  0xc710e1c8  0x57409b84  0x00000009
0xc72bdf78: 0xc72bdfb4  0xc72bdf7c  0xc72bdf7c  0xc72bdef4
0xc72bdf88: 0xc017b72e  0xc72bdf8c  0xc72bdf8c  0xc72bdf94
0xc72bdf98: 0xc72bdf94  0x00000000  0x00000000  0xcf91fe00

fp2
0xcb1d5088: 0x00000000  0x00000010  0x00000001  0xdead4ead
0xcb1d5098: 0xffffffff  0xffffffff  0x00000000  0x00000000
0xcb1d50a8: 0x00200200  0xcb80ebc8  0xe50e3f24  0x0000000a
0xcb1d50b8: 0xcb1d50f4  0xcb1d50bc  0xcb1d50bc  0xcb1dcf84
0xcb1d50c8: 0xc017b72e  0xcb1d50cc  0xcb1d50cc  0xcb1d50d4
0xcb1d50d8: 0xcb1d50d4  0x026a0005  0x00000000  0xcf91fe00

true
0xc001c0a8: 0x00000000  0x00000000  0x00000001  0xdead4ead
0xc001c0b8: 0xffffffff  0xffffffff  0x00000000  0xc67617f4
0xc001c0c8: 0xc12a0e7c  0xc727faa8  0xbfbb9195  0x00000009
0xc001c0d8: 0xc001c114  0xc001c16c  0xc05b9f5c  0xc001c174
0xc001c0e8: 0xc727faec  0xc001c0ec  0xc001c0ec  0xc001c0f4
0xc001c0f8: 0xc001c0f4  0x8bffffff  0x00000000  0xcf91fe00
```

(a) False Positive of `dentry`

```
struct sysfs_dirent {
   [0] atomic_t s_count;
   [4] struct list_head s_sibling;
  [12] struct list_head s_children;
  [20] void *s_element;
  [24] int s_type;
  [28] umode_t s_mode;
  [32] struct dentry *s_dentry;   [pruned]
  [36] struct iattr *s_iattr;     [pruned]
  [40] atomic_t s_event; }

fp1
0xcffaeffc: 0x00000000  0xcffa3800  0xcffaf800  0xcffa3808
0xcffaf00c: 0xcffaf808  0xcffc2800  0x00000000  0x00000000
0xcffaf01c: 0xcfd9bde0  0x00000008  0x70008086

fp2
0xcffaf7fc: 0x00000000  0xcffaf000  0xc03709a8  0xcffaf008
0xcffaf80c: 0xcffc2814  0xcffc2800  0x00000000  0x00000000
0xcffaf81c: 0xcfd9be60  0x00000000  0x12378086

fp3
0xcffa37fc: 0x00000000  0xcffa3000  0xcffaf000  0xcffa3008
0xcffa380c: 0xcffaf008  0xcffc2800  0x00000000  0x00000000
0xcffa381c: 0xcfd9bd60  0x00000009  0x70108086

fp4
0xcffa2ffc: 0x00000000  0xcffa2800  0xcffa3800  0xcffa2808
0xcffa300c: 0xcffa3808  0xcffc2800  0x00000000  0x00000000
0xcffa301c: 0xcfd9bce0  0x0000000b  0x71138086

fp5
0xcffa27fc: 0x00000000  0xcffa2000  0xcffa3000  0xcffa2008
0xcffa280c: 0xcffa3008  0xcffc2800  0x00000000  0x00000000
0xcffa281c: 0xcfd9bc60  0x00000010  0x00b81013

fp6
0xc037099c: 0x00000000  0xcffc2800  0xcffc2800  0xcffaf800
0xc03709ac: 0xcffa2000  0xc0327d79  0x00000000  0x00000124
0xc03709bc: 0xc01de4bc  0x00000000  0x00000000
```

(b) False Positive of `sysfs_dirent`

Figure 9: False Positive Analysis

we carefully check each field value, especially the boxed ones: the 0xdead4ead (SPINLOCK MAGIC at offset 12) and 0xcf91fe00 (a pointer to `dentry_operations` at offset 88), it is hard to believe these are not `dentry` instances, when compared with the true instance. As such, we suspect they are not FPs, and they are the cases that the slab allocator has freed the memory page of the destroyed `dentry` instances.

We have 6 FPs in `sysfs_dirent` data structure among the 2116 detected instances. The detailed memory dump of these 6 FP cases is shown in Figure 9(b). After our dynamic refinement, the fields at offsets 32 and 36 are pruned because they often contain `null` pointers, and the final signature entails checking two `list_head` data structures followed by a `void*` pointer at offset 4, 8, 12, 16 and 20, and four non-pointer field checking. Note one `list_head` has only two fields: previous and next pointer. However, there are 6 memory chunks that match our signature in the testing image. The chunks are not captured as part of the ground truth of any data structures. We suspect that it could be the case that they are aggregations of multiple data structures and the aggregations coincidentally manifest the pattern.

Appendix III: Performance Overhead

We also measured the performance overhead of our scanner (i.e., the parser). We run both our scanner and value invariant scanners on the testing image (256MB) in a machine with 3GB memory and an Intel Core 2 Quad CPU (2.4Ghz) running Ubuntu-9.04 (Linux kernel 2.6.28-17). The final result of the normalize performance overhead is shown in Figure 10. We could see the performance overhead for our scanner is still acceptable. We need to do address translation when there is a memory de-reference, but there is no need in value-invariant scanner. Thus, in all the case value-invariant inevitably performs better than our scanner. If the depth is relatively small, such as the 10 case with depth $D = 2$, our scanner only has 10X to 20X overhead than value invariant scanner. The deeper, the worse in our scanner, because more nodes need to be examined and more address translation needs to be involved, this is why the case of `inode_security` (with $D = 7$) and `mb_cache_entry` (with $D = 6$), we have big performance overhead. Thus, if the depth is not so high for the desired data structure, our system may be used as an online scanner. For example, in our experiment, it actually only takes a few seconds when scanning `fs_struct`, `thread_info`, and `files_struct`.

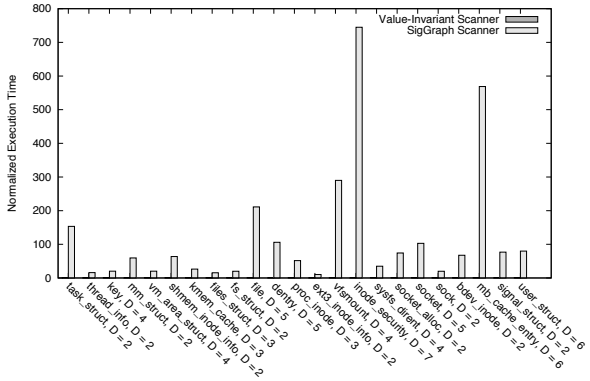


Figure 10: Performance Overhead