SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures

Zhiqiang Lin[†], Junghwan Rhee[†], Xiangyu Zhang[†], Dongyan Xu[†], Xuxian Jiang[‡]
[†]Dept. of Computer Science and CERIAS, Purdue University
{zlin, rhee, xyzhang, dxu}@cs.purdue.edu
[‡]Dept. of Computer Science, North Carolina State University
jiang@cs.ncsu.edu

Abstract

Brute force scanning of kernel memory images for finding kernel data structure instances is an important function in many computer security and forensics applications. Brute force scanning requires effective, robust signatures of kernel data structures. Existing approaches often use the value invariants of certain fields as data structure signatures. However, they do not fully exploit the rich points-to relations between kernel data structures. In this paper, we show that such points-to relations can be leveraged to generate graph-based structural *invariant signatures. More specifically, we develop SigGraph, a framework that systematically generates* non-isomorphic *signatures for data structures in an OS kernel. Each signature is a graph rooted at a subject data structure with its edges reflecting the points-to relations with other data structures. Our experiments with a range of Linux kernels show that SigGraph-based signatures achieve high accuracy in recognizing kernel data structure instances via brute force scanning. We further show that SigGraph achieves better robustness against pointer value anomalies and corruptions, without requiring global memory mapping and object reachability. We demonstrate that SigGraph can be applied to kernel memory forensics, kernel rootkit detection, and kernel version inference.*

1 Introduction

Given a kernel data structure definition, identifying instances of that data structure in a kernel memory image is an important capability in memory image forensics [28, 11, 22, 37, 34], kernel integrity checking [26, 10, 13, 27, 8], and virtual machine introspection [15, 18, 25]. Many state-of-the-art solutions rely on the *field value invariant* exhibited by a data structure (i.e., a field with either constant value or value in a fixed range) as its signature [38, 35, 13, 9, 8]. Unfortunately, many kernel data structures cannot be covered by the value-invariant scheme. For example, some data structures do not have fields with invariant values or value ranges. It is also possible that an invariant-value field be corrupted, 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.

We present a complementary scheme 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 its 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 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 data structures, the "topology" of kernel data structures (formed by "points-to" relations) is more stable. As such, SigGraph has the promise to uniquely identify kernel data structures with pointers.

A salient feature of SigGraph-based signatures is that they can be used for *brute force scanning*: Given an *arbitrary* kernel memory address x, a signature (more precisely, a memory scanner based on it) can decide if an instance of the corresponding data structure exists in the memory region starting at x. As such, SigGraph is different from the global "top-down" scanning employed by many memory mapping techniques (e.g., the ones for software debugging [30] and kernel integrity checking [26, 10]). Global "top-down" scanning is enabled by building a global points-to graph for a subject program – rooted at its global variables and expanding to its entire address space. Instances of the program's data structures can then be identified by traversing the global graph starting from the root. On the other hand, brute force scanning is based on multiple, context-free points-to graphs – each rooted at a distinct data structure. Unlike global scanning, brute force scanning

does not require that a data structure instance be "reachable" from a global variable in order to be recognized, hence achieving higher robustness against attacks that tamper with such global reachability (an example of such attack is presented in Section 8.1).

To enable brute force scanning, SigGraph faces the new challenge of *data structure isomorphism*: The signatures of different data structures, if not judiciously determined, may be *isomorphic*, leading to false positives in data structure instance recognition. To address this challenge, we formally define data structure isomorphism and develop an algorithm to compute unique, non-isomorphic signatures for kernel data structures. From the signatures, data structure-specific *kernel memory scanners* are automatically generated using context-free grammars. To improve the practicality of our solution, we propose a number of heuristics to handle practical issues (e.g., some pointers being `null`). Interestingly, we obtain two important observations when developing SigGraph: (1) The wealth of points-to relations between kernel data structures allows us to generate *multiple* signatures for the same data structure. This is particularly powerful when operating under malicious pointer mutation attacks, thus raising the bar to evade SigGraph. (2) The rich points-to relations also allow us to avoid complex, expensive points-to analysis of kernel source code for `void` pointer handling (e.g., as proposed in [10]). Distinct data structure signatures can be generated *without* involving the generic pointers.

We have performed extensive evaluation on SigGraph-based signatures with several Linux kernels and verified the uniqueness of the signatures. Our signatures achieve low false positives and zero false negatives when applied to data structure instance recognition in kernel memory images. Furthermore, our experiments show that SigGraph works without global memory maps and in the face of a range of kernel attacks that manipulate pointer fields, demonstrating its applicability to kernel rootkit detection. Finally, we show that SigGraph can also be used to determine the version of a guest OS kernel, a key pre-requisite of virtual machine introspection.

2 Overview

2.1 Problem Statement and Challenges

SigGraph exploits the inter-data structure points-to relations to generate non-isomorphic data structure signatures. Consider seven simplified Linux kernel data structures, four of which are shown in Figure 1(a)-(d). In particular, `task_struct(TS)` contains four pointers to `thread_info(TI)`, `mm_struct(MS)`, `linux_binfmt(LB)`, and `TS`, respectively. `TI` has a pointer to `TS` whereas `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

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

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 ("bottom-up") tends to be true. In other words, we can use the graph as the signature of `task_struct` and perform the *reverse* inference as follows.

$$\begin{aligned} S_{\texttt{TS}}(x) \leftarrow \quad & S_{\texttt{TS}}(*(*(x+0)+0)) \ \wedge \ S_{\texttt{VA}}(*(*(x+4)+0)) \wedge \\ & S_{\texttt{FP}}(*(*(x+4)+4)) \ \wedge \ S_{\texttt{MD}}(*(*(x+8)+0))) \\ & \wedge S_{\texttt{TS}}(*(x+12)) \end{aligned} \tag{6}$$

Different from the global memory mapping techniques (e.g., [30, 26, 10, 28, 34, 11, 22]) SigGraph aims at deriving unique signatures for *individual* data structures for brute force kernel memory scanning. Hence we face the following new challenges:

- **Avoiding signature isomorphism:** 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 new challenge is to identify the sufficient and necessary conditions to avoid signature isomorphism between data structures.

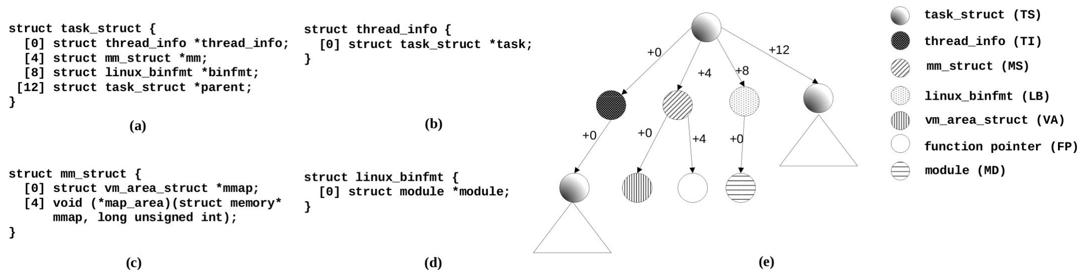


Figure 1. A working example of kernel data structures and a graph-based data structure signature. The triangles indicate recursive definitions

- **Generating signatures:** Meanwhile it is possible that one 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 scanner 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

An 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) scanner generator. To generate signatures, SigGraph first extracts data structure definitions from the OS source code. This is done automatically through a compiler pass (Section 3). To handle practical issues such as `null` pointers and `void*` pointers, the *profiler* identifies problematic pointer fields via dynamic analysis (Section 6). The *signature generator* checks if non-isomorphic signatures exist for the data structures and if so, generates such signatures (Section 4). The generated signatures are then automatically converted to the corresponding kernel memory scanners (Section 5), which are the "product" shipped to users. A user will simply run these *scanners* to perform brute-force scanning over a kernel memory image (either memory dump or live memory), with the output being the instances of the data structures in the image.

3 Data Structure Definition Extraction

SigGraph's *data structure definition extractor* adopts a compiler-based approach, where the compiler pass is devised to walk through the source code and extract data structure definitions. It is robust as it is based on a full-fledged language front-end. In particular, our development is in `gcc-4.2.4`. The compiler 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 output of the compiler pass is the data structure definitions – with offset and type for each field – 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 debugging information is provided along with the binary, SigGraph can simply leverage the debugging information to extract the data structure definitions. Otherwise, data structure reverse engineering techniques (e.g., REWARDS [21] and TIE [20]) can be leveraged to extract data structure definitions from binaries.

4 Signature Generation

Suppose 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 starting at address x is an instance of t. The following production rule can be generated for T:

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

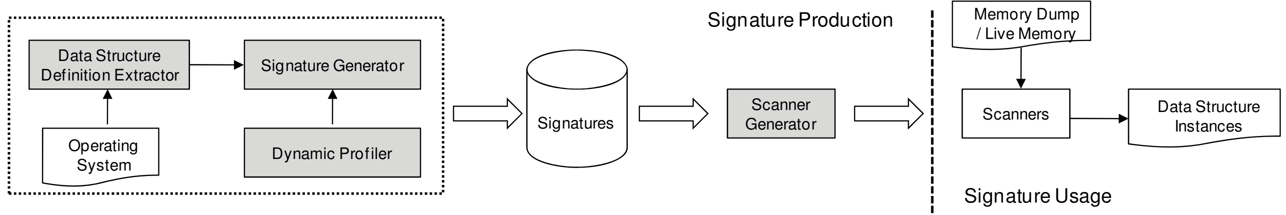


Figure 2. SigGraph system overview

Brute force memory scanning is based on the *reverse* of the above rule: Given a kernel memory image, 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 *any* 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 will be the root of the graph. For convenience of discussion, we assume for now that pointers are not `null` and they each have an explicit type (i.e., not a `void` pointer). We will address the cases where this assumption does not hold in Section 6.

As noted 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 that data structure.* Consider Figure 3(a), data structures `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 structures as their signatures. However, this is not true. Consider the memory segment at the bottom of Figure 3(a), 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 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 isomorphism 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 being any pointer type.

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, disregarding 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(a), $IPP(A) = 0{\cdot}B{\cdot}12{\cdot}C{\cdot}6{\cdot}D$ 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 3(b).

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

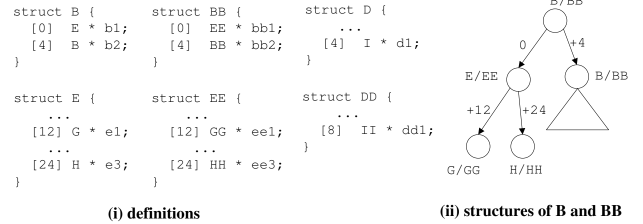
Strings (1) and (2) above are both pointer patterns of A. The pointer patterns of a data structure are candidates for its signature. As one data structure may have many pointer patterns, the challenge becomes to algorithmically identify the unique pointer patterns of a given data structure so that instances 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 be identified.

```
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
```

(a) Insufficiency of pointer layout uniqueness



(i) definitions **(ii) structures of B and BB**

(b) Data structure isomorphism

Figure 3. Examples illustrating the signature isomorphism problem

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 structures 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 3(b) (i) shows the definitions of some data structures in Figure 3(a). 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 substructures ($E \bowtie EE$) 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 cannot be generated from any other individual data structure through expansions.

The proof of Theorem 1 is omitted for brevity. 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 conditions <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 all data structures). Fortunately, most of those are data structures that are rarely used or not widely targeted according to OS security and forensics literature.

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

Consider the data structures in Figure 3(a) and Figure 3(b). 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 because of their different immediate pointer patterns. According to Theorem 1, there must be a unique signature for A. In this example, pointer pattern (2) in Equation (8) is a unique signature. 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 using Theorem 1, there may be multiple pointer patterns of T that can

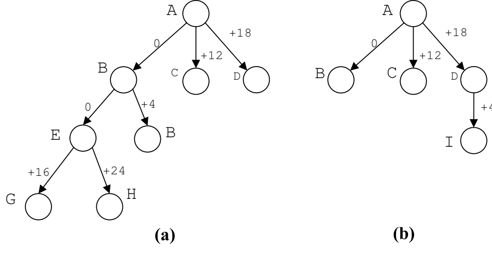


Figure 4. If the offset of field `e1` (of type `struct G`) in `E` is changed to 16, `struct A` will have two possible signatures (detailed data structure definitions in Figure 3)

distinguish T from other data structures. Ideally, we want to find the minimal pattern as it incurs the minimal parsing overhead during brute force scanning. For example, if the offset of field `e1` (of type `struct G`) in `E` is 16, `struct A` will have two possible signatures as shown in Figure 4. They correspond to the following 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

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

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

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 cannot 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 (Algorithm 1) 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 search algorithm that performs expansions for all pointer symbols at 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 set $distinct$. 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(a) and 3(b), the pattern generated for A by the algorithm is

$$0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I) \quad (9)$$

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

Generating Multiple Signatures. In some use scenarios, it is highly desirable to generate *multiple* signatures for the same data structure. A common scenario is that some

Algorithm 1 An approximate algorithm for signature generation

Input: Data structure T and set K of all kernel data structures considered
Output: The pointer pattern that serves as the signature of T.
1: s= $IPP(T)$
2: **let** $IPP(T)$ be $f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot \ldots \cdot (f_n - f_{n-1}) \cdot t_n$
3: **for** each sub-pattern p=$g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot \ldots \cdot (f_n - f_{n-1}) \cdot r_n$ in $IPP(R)$ of each structure $R \in (K - \{T\})$ 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 disregarding 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

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 for the same data structure. In particular, if certain pointer fields in a data structure are potential targets of malicious manipulation, SigGraph will *avoid* using such fields during signature generation in Algorithm 1. For example, if field `e1`'s offset in `struct E` is 16 and field `a3` (of type `struct D`) in `struct A` is not dependable, Algorithm 1 will adapt (not shown in the pseudo-code) by *pruning* the sub-graph rooted at field `a3` in Figure 4(a).

5 Scanner Generation

Given a data structure signature (i.e., a pointer pattern), SigGraph will automatically generate the corresponding memory scanner, which will be shipped to users for brute force kernel memory scanning. To automatically generate scanners, we describe all signatures using a context-free grammar (CFG). Then we leverage `yacc` to generate the scanners. The CFG is described as follows.

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

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 see that the grammar describes all the pointer patterns in Section 4, such as the signature of A generated by Algorithm 1 (Equation (9)).

Scanners 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 scanner asserts that the corresponding memory contain a pointer. If a '(' symbol is encountered, a pointer dereference is performed and the scanner starts to parse the next-level memory region until the matching ')' is encountered. A sample scanner generated for the signature in Equation (9) can be found in Figure 5. Function `isInstanceOf_A` decides if a given address is an instance of A; `assertPointer` asserts that the given address must contain a pointer value, otherwise an exception will be thrown and function `isInstranceOf_A` will return 0. The `yacc` rules to generate scanners are elided for brevity.

Considering Non-pointer Fields. So far, a scanner considers only the positive information from the signature, which indicates the fields that are supposed to be pointers. But it does not consider the implicit negative information, which indicates the fields that are supposed to be non-pointers. In many cases, such negative information is needed to construct robust scanners.

For example, assume that 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 T's signature is unique and has no structural conflict with any other data structures, the scanner of T will mistakenly identify part of the array as an instance of T.

To handle such cases, the scanner should also assert that the non-pointer fields must not contain pointers. Hence the scanner for T's signature becomes the following. Method `assertNonPointer` asserts that the given address does not contain a pointer. As such, the final scanner code for identify data structure T will be:

```
1 int isInstanceOf_T(void *x){
2   x=x+0;
3   assertPointer(*x);    // field of type "A *"
4   x=x+4;
5   assertNonPointer(*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

We have so far assumed the ideal case 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 solutions to the following key problems.

1. **Null pointers:** It is possible that a pointer field have a `null` value, which cannot be distinguished from other non-pointer fields, such as integer or floating point fields with value 0. If 0 is considered a pointer value, a memory region with all 0s 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 case.

3. **User-level pointers:** It is also possible that a kernel pointer point to the user space. For example, the `set_child_tid` and `clear_child_tid` fields in `task_struct`, and the `vdso` field in `mm_struct` all point to user space. The difficulty is that user space pointers have a very dynamic value range due to the larger 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_POISON2` with value `0x00200200` as two special pointers to verify that no one uses un-initialized list entries. Another special value `SPINLOCK_MAGIC` (`0xdead4ead`) also widely exists 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 an 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 when pointer fields are involved.

7. **Rarely accessed data structures:** Algorithm 1 in Section 4 treats all data structures equally and tries to find unique signatures for all kernel 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 problems above boil down to the difficulty in deciding if a field is pointer or non-pointer. Interestingly, 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 (Figure 2) relies on LiveDM [33], a dynamic kernel memory mapping system, to keep track of dynamic kernel data structures at runtime. Based on QEMU [3], LiveDM tracks kernel memory allocation and deallocation events. More specifically, we focus on slab objects by hooking the allocation and deallocation functions such as `kmem_cache_alloc` and `kmem_cache_free` at the

```
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 5. The generated scanner for `struct A`**'s signature in Equation (9)**

VMM level. The function arguments and return values are retrieved to obtain memory ranges of these objects. Their types are acquired by mapping allocation call sites to kernel data types via static analysis. We then track the life time of these objects and monitor their values.

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 experiments, we profile a number of kernel executions for long periods of time (hours to tens of hours).

Based on 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 declared undependable (Section 4), which is done by annotating them with a special symbol. Note that they should *not* be considered as non-pointer fields either and 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[1] 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 such as `0x00100100` or `0x00200200`.

We point out that dynamic profiling and signature refinement is performed only during the *production* of SigGraph-based signatures/scanners. It is *not* performed by end-users, who will simply run the scanners on memory images. We do note that the SigGraph signatures/scanners are kernel-specific, as different OS kernels may have different data structure definitions and runtime access characteristics. In fact, Section 8.2 shows that different versions of the same OS kernel may have different signatures for the same data structure.

[1]We note that such exclusion will *not* remove important pointer fields in critical kernel data structures such as lists and trees, where non-zero magic values are used to indicate list/tree termination or initialization.

7 Evaluation

We have implemented a prototype of SigGraph in C and Python. More specifically, we instrument `gcc-4.2.4` to traverse ASTs and collect kernel data structure definitions. Our scanner generator is lex/yacc based, and the generated scanners are in C. The entire implementation has around 10K lines of C code and 6K lines of Python code.

7.1 Signature Uniqueness

We first test if unique signatures exist for kernel data structures. We test 5 popular Linux distributions (from Fedora Core 5 and 6; and Ubuntu 7.04, 8.04 and 9.10), with the corresponding kernel version shown in the first column of Table 1. 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 5th column) of the data structures with pointer fields have unique signatures. Because of graph isomorphism, there are data structures that do not have any unique signature, and the percentage for these data structures is around 12%. For the average steps ($\overline{S}$) performed in pointer pattern expansion to generate the unique signatures, the numbers are shown in the 6th column. Note that these are all static numbers before the dynamic refinement.

From the 7th to the 20th column in Table 1, we show the number of unique signatures of various depths, obtained by taking various number of expansion steps along the points-to relations. 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 Signature Effectiveness

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 choose 23 widely used kernel data structures shown in the 2nd column of Table 2. We choose these data structures because: (1) They are the most commonly examined data structures in existing literature [28, 11, 22, 37, 34, 38, 35, 9]; (2) They are important data structures that can represent

Kernel version	#Total structs	Signature Statistics				Number of Signatures of Various Depths												
		#Pointer structs	#Unique Sig.	Percent	$\overline{S}$	1	2	3	4	5	6	7	8	9	10	11	12	13
2.6.15-1	8850	3597	3229	89.76%	2.31	1355	823	461	229	76	194	85	4	1	0	1	-	-
2.6.18-1	11800	4882	4305	88.18%	2.45	1820	1057	382	410	159	337	121	9	3	5	1	1	-
2.6.20-15	14992	6096	5395	88.50%	2.54	2137	1311	680	236	407	501	106	9	1	5	1	1	-
2.6.24-26	15901	6427	5645	87.83%	2.47	2172	1316	761	475	624	248	37	7	1	0	3	1	-
2.6.31-1	26799	9957	8683	87.20%	2.73	3364	1951	696	319	1492	494	344	19	1	0	1	1	1

Table 1. Experimental results of signature uniqueness test

the status of the system in the aspects of process, memory, network and file system; from these data structures, we can reach most other kernel objects; and (3) They contain pointer fields. Note that when scanning for instances of these data structures, other data structures – as part of the pointer patterns – are also traversed.

To ease our presentation, we assign an ID to each data structure, which is shown in the 3rd 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 5th and 6th columns, respectively.

7.2.1 Experiment Setup

We perform two sets of experiments. We first use our profiler to automatically prune the undependable pointer/non-pointer fields, generate refined signatures, and then detect the instances. After that we perform a comparison run with value invariant-based signatures (Section 7.2.3) to further confirm the effectiveness of SigGraph.

Memory snapshot collection: The first input of 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 study. 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 our Python script. Note that to enable `crash`'s dump analysis, the kernel needs to be rebuilt with debugging information.

Profiling run: In all our profiling runs, the OS kernel is executed under normal workload and monitored for hours, with the goal of achieving good coverage of kernel data access patterns. However, it is unlikely that the profiling runs be able to capture the complete spectrum of patterns. As our future work, we will leverage existing techniques for software test generation to achieve better coverage.

7.2.2 Dynamic Refinement

In this experiment, we carry out the dynamic refinement phase as described in Section 6. The depth and size of signatures before and after pruning are presented in the "SigGraph Signature" columns in Table 2, with D being 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 SigGraph-based signatures with value invariant-based signatures [38, 35, 13, 9], we also implement a basic value-invariant signature generation system. More specifically, we generally derive four types of invariants for each field including (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 such value invariants for the data structures, we perform two types of profiling: one is access frequency profiling (to prune out the fields that are never accessed by the kernel) and the other is to sample their values and produce the signatures. The access frequency profiling is done by instrumenting QEMU to track memory reads and writes. Sampling is similar to the sampling method in our dynamic refinement phase.

All the data structures under study turn out to have value invariants. The statistics of these signatures are shown 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 brute force scanning a test image is shown in Table 3. The 3rd column shows the total number of true instances of the data structure, which is acquired by the modified `crash` utility [2].

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 for Linux kernel 2.6.18-1

The $|R|$ column shows the number of data structure instances detected by the scanning. Due to the limitation of crash, the ground-truth instances are live, namely reachable from global or stack variables. On the other hand, brute force scanning can further identify freed-but-not-yet-reallocated objects that are not reachable from global or stack variables. Such freed objects detected would be counted as false positives (FPs) when compared with the ground truth from crash. As such, we present two FP numbers: (1) $|FP'|$ for those false positives that include the freed objects and (2) $|FP|$ for those that do not include the freed objects (hence $|FP'| \geq |FP|$). The false negative FN indicates those missed by scanning but present among the ground truth objects from crash.

Among the 23 data structures, SigGraph perfectly (namely with accuracy and completeness) identifies all instances of 16 of the data structures when freed objects are considered FPs (i.e., both FP' and FN are zero); whereas value invariant signatures perfectly identify only 5 of the data structures. When freed objects are not considered FPs, 20 data structures can be perfectly identified by SigGraph whereas value invariant signatures perfectly identify 9. We also note that, with the exception of dentry, SigGraph signatures achieve equal or (much) lower false positive rate than value invariant-based signatures. No FNs are observed for SigGraph, while some are observed for the value invariant-based approach.

False Positive Analysis. Table 3 shows that SigGraph results in false positives ($|FP|$) for three of the 23 data structures: vm_area_struct, dentry, and sysfs_dirent. We carefully examine the memory snapshot and identify the reasons as follows.

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

- **vm_area_struct** We have 9 false positives (FPs) among the 2233 detected instances. After dynamic refinement, some pointer fields are pruned, such as the pointer field at offset 12 (as shown in Figure 6). The resultant signature consists of a pointer field at offset 0 (mm_struct), followed by a sequence of non-pointer fields, and so on. However, field task_struct starting from offset 156 has the same pointer pattern as that of vm_area_struct *except* that offset 160 is a pointer. Unfortunately, in some rare cases that are *not* captured by our profiler, the pointer field at offset 160 becomes 0, leading to the 9 FPs.

- **dentry** We have 2 FPs of dentry, which are shown in Figure 7(a). We consider these two instances as FPs because they cannot be found in ei-

ID	Data Structure Name	$\|I\|$	SigGraph Signature				Value Invariant Signature			
			$\|R\|$	FP'	FP	FN	$\|R\|$	FP'	FP	FN
1	task_struct	88	88	0.00%	0.00%	0.00%	88	0.00%	0.00%	0.00%
2	thread_info	88	88	0.00%	0.00%	0.00%	93	6.45%	6.45%	1.08%
3	key	22	22	0.00%	0.00%	0.00%	19	0.00%	0.00%	15.79%
4	mm_struct	52	54	3.70%	0.00%	0.00%	55	5.45%	0.00%	0.00%
5	vm_area_struct	2174	2233	2.64%	0.40%	0.00%	2405	9.61%	7.52%	0.00%
6	shmem_inode_info	232	232	0.00%	0.00%	0.00%	226	0.00%	0.00%	2.65%
7	kmem_cache	127	127	0.00%	0.00%	0.00%	5124	97.52%	97.52%	0.00%
8	files_struct	53	53	0.00%	0.00%	0.00%	50	0.00%	0.00%	6.00%
9	fs_struct	52	60	13.33%	0.00%	0.00%	60	13.33%	0.00%	0.00%
10	file	791	791	0.00%	0.00%	0.00%	791	0.00%	0.00%	0.00%
11	dentry	31816	38611	17.60%	0.01%	0.00%	31816	0.00%	0.00%	0.00%
12	proc_inode	885	885	0.00%	0.00%	0.00%	470	0.00%	0.00%	88.30%
13	ext3_inode_info	38153	38153	0.00%	0.00%	0.00%	38153	0.00%	0.00%	0.00%
14	vfsmount	28	28	0.00%	0.00%	0.00%	28	0.00%	0.00%	0.00%
15	inode_security	40067	40365	0.74%	0.00%	0.00%	142290	71.84%	70.93%	0.00%
16	sysfs_dirent	2105	2116	0.52%	0.52%	0.00%	88823	97.63%	97.63%	0.00%
17	socket_alloc	75	75	0.00%	0.00%	0.00%	75	0.00%	0.00%	0.00%
18	socket	55	55	0.00%	0.00%	0.00%	49	0.00%	0.00%	12.24%
19	sock	55	55	0.00%	0.00%	0.00%	43	0.00%	0.00%	27.90%
20	bdev_inode	25	25	0.00%	0.00%	0.00%	24	0.00%	0.00%	4.17%
21	mb_cache_entry	520	633	17.85%	0.00%	0.00%	638	18.50%	0.00%	0.00%
22	signal_struct	73	73	0.00%	0.00%	0.00%	72	0.00%	0.00%	1.39%
23	user_struct	10	10	0.00%	0.00%	0.00%	10591	99.91%	99.91%	0.00%

Table 3. Experimental results of SigGraph signatures and value invariant-based signatures

ther the pool of live objects or the pool of freed objects. However, if we carefully check each field's value, especially the boxed ones: 0xdead4ead (SPINLOCK_MAGIC at offset 12) and 0xcf91fe00 (a pointer to dentry_operations at offset 88), we cannot help but thinking that these are indeed dentry instances instead of FPs. We believe that they belong to the case where the slab allocator has freed the memory page of the destroyed dentry instances.

- **sysfs_dirent** We have 6 FPs of sysfs_dirent among the 2116 detected instances. The detailed memory dumps of the 6 FP cases are shown in Figure 7(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 offsets 4, 8, 12, 16 and 20, respectively) and checking four non-pointer fields. Note that each list_head has only two fields: previous and next pointer. There are 6 memory chunks that match our signature in the test memory image. But the chunks are not part of the ground truth. We suspect that these chunks are *aggregations* of multiple data structures and the aggregations coincidentally manifest the same pattern.

Summary: In this experiment, SigGraph achieves zero FN and (much) lower FP rates. Intuitively, the reasons are the following: (1) SigGraph-base signatures are structure-oriented and thus tend to be more stable than value-oriented approaches. And their uniqueness can be algorithmically determined – that is, we can expand a signature along available points-to edges to achieve uniqueness. (2) SigGraph-based signatures are more "informative" as each signature includes information about *other* data structures; whereas a value-based signature only carries information about itself.

7.3 Multiple Signatures

One powerful feature of SigGraph is that multiple signatures can be generated for the same data structure (Section 4). We perform the following experiments with the task_struct data structure to verify that. In each experiment, we exclude one of the 38 pointer fields of task_struct (considering that pointer corrupted) before running Algorithm 1. In each of the 38 experiments, 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 $\binom{2}{38}$ 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.

The above experiments indicate that SigGraph is robust in the face of corrupted pointer fields. However, the robustness does have its limit. At the other extreme, we exclude 37 of the 38 pointer fields of task_struct and conduct $\binom{37}{38} = 38$ 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.

7.4 Performance Overhead

Since SigGraph may be used for online live memory analysis, we measure the overhead of memory scanning using SigGraph signatures. We run both SigGraph-generated

```
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  0x8bfffff9  0x00000000  0xcf91fe00
```

(a) False positives 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 positives of `sysfs_dirent`

Figure 7. False positive analysis of `dentry` and `sysfs_dirent`

scanners and the value invariant-based scanners on the testing image (256MB) in a machine with 3GB RAM and an Intel Core 2 Quad CPU (2.4GHz) running Ubuntu-9.04 (Linux kernel 2.6.28-17). The final result of the normalized overhead is shown in Figure 8.

As expected, value-invariant scanners always outperform SigGraph scanners. The main reason is that: A SigGraph scanner needs to conduct address translation whenever there is a memory de-reference, which is not needed by the value invariant scanner. If the depth of a SigGraph signature is relatively low (e.g., $D = 2$), the SigGraph scanner will be roughly 10-20 times slower than the corresponding value invariant scanner. Greater depth often leads to higher overhead because more nodes will need to be examined and more address translation needs to be performed. The cases of `inode_security` ($D = 7$) and `mb_cache_entry` ($D = 6$) are such examples. Thus, for data structures with low-depth signatures, their SigGraph scanners can be used online. For example, in our experiment, it takes only a few seconds to scan `fs_struct`, `thread_info`, and `files_struct`, and less than one minute to scan `task_struct`.

For data structures with a greater depth (due to isomorphism elimination) such as `inode_security` and `mb_cache_entry`, the scanning time is longer (e.g., about 15 minutes when we scan a 256MB memory image using the scanner for `inode_security`). However, we argue that such cost is acceptable in the context of computer forensics, where accuracy and completeness is more important than efficiency. Moreover, the scanning time can be reduced by various optimizations such as parallelization or having a pre-scanning phase to preclude unlikely cases.

8 Security Applications

SigGraph is naturally applicable to memory image analysis/forensics. Besides, we have applied SigGraph to two other security applications: kernel rootkit detection and kernel version inference.

8.1 Kernel Rootkit Detection

By uncovering the kernel objects in a kernel memory image, SigGraph provides the semantic view of kernel memory for kernel rootkit detection. We note the convenience of using SigGraph: The user simply runs the data structure-specific scanners on a subject memory image to uncover kernel objects of interest.

Based on the kernel objects revealed by SigGraph, we then follow the existing "view comparison" methodology [10, 18, 33] for kernel rootkit detection: For a certain type of kernel object (data structure), we compare (1) the number and values of its instances revealed by SigGraph with (2) the relevant information returned by a corresponding system utility (e.g., `lsmod` and `ps` for kernel modules and processes, respectively). If a discrepancy between the two

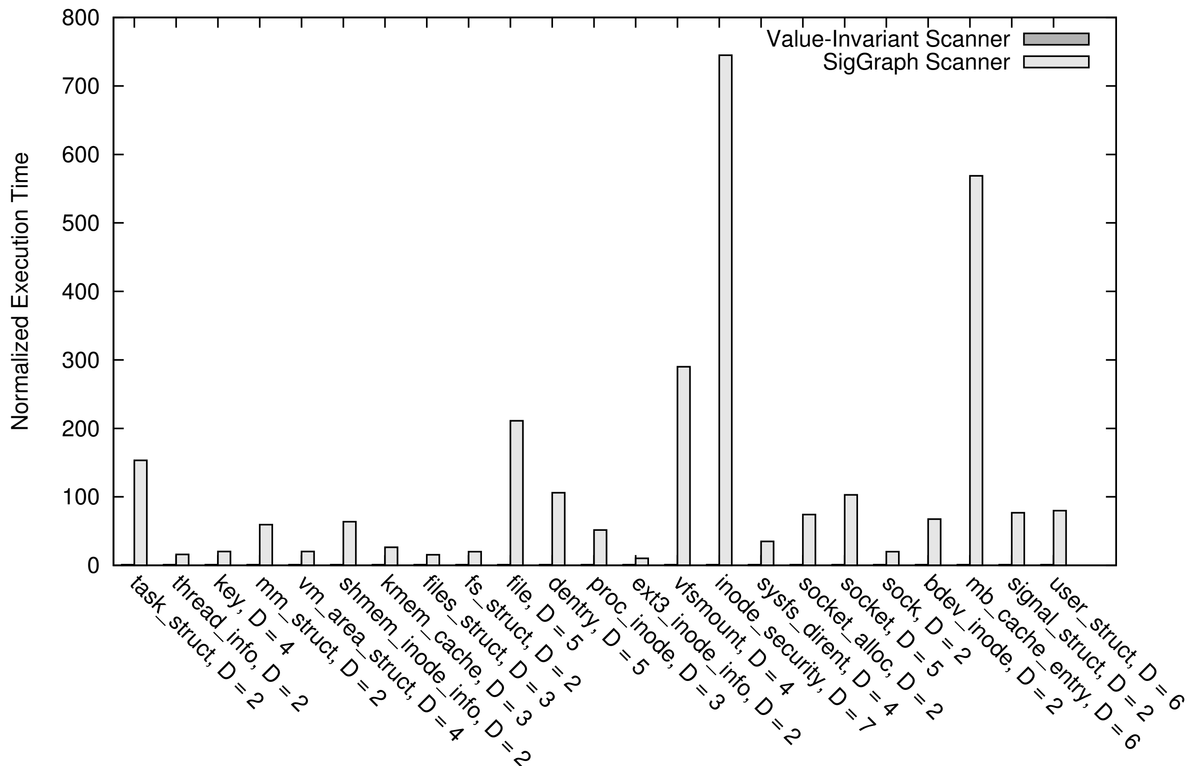


Figure 8. Memory scanning performance

views is observed, we know that certain kernel object(s) is being hidden, indicating a kernel rootkit attack.

Many kernel rootkits engage in kernel data hiding attacks [10, 18, 33]. Among those, we experiment with 8 representative real-world kernel rootkits (which cover the spectrum of data hiding techniques) and the results are presented in the first 8 rows in Table 4. SigGraph helps detect all of them. Specifically, we use the original samples of `adore-ng-2.6`, `adore-ng-2.6`', `override`, `enyelkm 1.0` and port those of `hp`, `linuxfu`, `modhide`, `cleaner` from Linux 2.4 to Linux 2.6 where our experiment is based on. All these rootkits except `adore-ng-2.6`' and `override` hide tasks or kernel modules by manipulating pointers. For example, `adore-ng` changes the connecting pointers of neighboring modules to hide its own module; and `enyelkm` calls a list function (`list_del`) that separates its own module from the module list. As a result, the number of the kernel modules counted by `lsmod` is *one less* than the number of corresponding kernel objects revealed by SigGraph, with the missing one being the rootkit module itself.

We point out that the success of kernel rootkit detection in these experiments is attributed to SigGraph's provision of *multiple alternative* signatures (Section 7.3) for the same data structure. With the kernel rootkit's presence, some pointers from/to a kernel object may be corrupted and can no longer be used for signature matching. For example, kernel modules are connected by `list.next` and `list.prev` pointers, which are manipulated by rootkits. Fortunately, SigGraph is able to generate alternative signatures that do not involve those pointers. With such signatures, SigGraph scanners accurately recognize the kernel objects that are being hidden.

Finally, rootkits `adore-ng-2.6`' and `override` have different attack mechanisms. They hide processes by filtering out information about the hidden processes using injected code – without manipulating kernel objects. SigGraph recognizes these objects using the default signature of `task_struct` without resorting to the alternative ones, which leads to the detection of such attacks via view comparison.

Comparison with techniques based on global memory mapping. A number of existing kernel rootkit detection techniques rely on building a graph that maps the *entire* live memory through pointers. The state of the art is KOP [10]. Based on Windows, it builds a global memory graph and resolves function pointers through an advanced points-to analysis. Due to the lack of its Linux implementation, we implement a basic system based on global memory mapping by extending the `crash` utility. As a core dump analysis infrastructure that resolves memory regions based on type information, `crash` is extendable for customized

Rootkit Name	Target Object	Inside view	crash		SigGraph		Description of the Rootkit Attack
		# of obj.s	# of obj.s	Detected	# of obj.s	Detected	
adore-ng-2.6	module	23	23	✗	24	✓	Hide its own module (self-hiding)
adore-ng-2.6'	task_struct	62	63	✓	63	✓	Hide one process using injected code
cleaner-2.6	module	22	22	✗	23	✓	Hide the next module of the rootkit
enyelkm 1.0	module	23	23	✗	24	✓	Hide its own module (self-hiding)
hp-2.6	task_struct	56	57	✓	57	✓	Hide one process with a given PID
linuxfu-2.6	task_struct	59	60	✓	60	✓	Hide one process with a given name
modhide-2.6	module	22	22	✗	23	✓	Hide one module with a given name
override	task_struct	58	59	✓	59	✓	Hide one process using injected code
rmroots	task_struct	56	N/A	✗	55	✓	Destroy static data structures to hide
rmroots'	module	23	N/A	✗	24	✓	Destroy static data structures to hide

Table 4. Experimental result on kernel rootkit detection

memory analysis. In particular, our extension involves a Python script to build a global memory graph by exploring the points-to relations. We consider a rootkit detected if the hidden kernel object (module or task) is reachable in the graph. Table 4 presents the results. The extended crash detects 4 out of the 8 real-world rootkits. It is not a surprise that crash detects adore-ng-2.6' and override as they do not manipulate kernel object pointers. For hp-2.6 and linuxfu-2.6, even though the rootkit tasks are hidden from the task list, they are still reachable via other data structures in the memory graph (more specifically via data structures for process scheduling). However, such alternative reachability is not available when running adore-ng-2.6, cleaner-2.6, enyelkm 1.0, and modhide-2.6 and hence crash misses them.

We note that global memory graph-based techniques rely on each object's reachability from the root(s) of the graph. In other words, an object cannot be properly typed if it is not reachable from the root(s). As a result, it is conceivable that future rootkits may try to destroy such reachability. For example, a rootkit may identify a *cut* of the global memory graph and destroy (or obfuscate) the pointers along the cut. Consequently, objects not reachable from the original roots become un-recognizable. As an extreme example, we construct two such rootkits: rmroots and rmroots' (the last two rows in Table 4). They hide task_struct and module instances, respectively and, to destroy evidence at the end of the attack, they "wipe out" the static kernel data structures listed in the kernel symbol table (system.map) so that the rest of the memory becomes un-mappable.

In comparison, SigGraph shows better robustness against such an attack. In our experiment with the rmroots rootkit, there are 56 running processes right before the static kernel object wipe-out. Soon after the wipe-out, the system crashes due to pointer corruption and a kernel memory snapshot is taken. We run the extended crash on the kernel memory image but it fails to construct the global memory graph due to the absence of static kernel objects. On the other hand, SigGraph is able to identify 55 instances of task_struct, *including* the one that was hidden before the wipe-out (The missing one is actually init_task, an instance of task_struct that has been cleared). For our experiment with rmroots', the SigGraph scanner successfully identifies all 24 kernel modules including the one being hidden.

8.2 Kernel Version Inference

Another application of SigGraph is the determination of OS kernel version based on a kernel memory snapshot. Consider the following scenario: A public cloud computing platform hosts virtual machines (VMs) with various OS kernels. In order to perform virtual machine introspection [15, 18, 25] on these guest VMs (e.g., for intrusion/malware detection and usage auditing), a prerequisite is to know the specific version of a guest's OS kernel [16, 36, 5]. The kernel type/version is critical to accurately interpreting the VM's system state and events by the VMM. However, such information is not always available to the cloud provider (e.g., the cloud provider only knows that a VM runs Linux but doesn't know which version).

Currently guest kernel version can be determined via value invariants (e.g., as adopted in [18]). We instead propose using SigGraph-based data structure signatures as a more accurate kernel version indicator. To validate our proposal, we take 9 more Linux kernels ending with an even version number from 2.6.12 to 2.6.34. We select this range because they all work with our gcc-4.2.4-based implementation. If a selected version has multiple sub versions, we take the latest one. Together with the 5 Linux kernels already tested (marked with *), we have a total of 14 kernel versions, which are listed in the 1st column of Table 5.

Version indicator selection. We first compile these kernels using the default configuration to get all their data structure definitions. We then derive SigGraph-based signatures for all data structures. After that we try to select one data structure whose signatures in different kernel versions can be used to differentiate the kernel versions. The main requirements for such a data structure D are: (1) It should be commonly present in the execution of all kernels; and (2) Its signatures should be distinctive across different kernels. In other words, for each kernel version i, we shall find a signature S_i of D that will recognize instances of D *in and only in* memory images of kernel version i. In the end, we are not able to find a single data structure that can differentiate all the 14 kernels due to the similarity among them. (In fact, we find that two of the kernels share the same data structure definitions.) However, we do find that

Linux kernel version	thread _info	process name	mm_struct		task_struct		list_head					Signature uniqueness?
			mm	active _mm	real_ parent	parent	tasks	ptrace_ children	ptrace _list	children	sibling	
2.6.12-6	4	436	108	112	152	156	84	92	100	160	168	✓
2.6.14-7	4	428	120	124	164	168	96	104	112	172	180	✗
2.6.15-1*	4	428	120	124	164	168	96	104	112	172	180	✗
2.6.16-62	4	432	120	124	164	168	96	104	112	172	180	✓
2.6.18-1*	4	428	152	156	196	200	128	136	144	204	212	✓
2.6.20-15*	4	404	128	132	172	176	104	112	120	180	188	✓
2.6.22-19	4	408	132	136	176	180	108	116	124	184	192	✓
2.6.24-26*	4	461	164	168	208	212	140	148	156	216	224	✓
2.6.26-8	4	505	188	192	232	236	164	172	180	240	248	✓
2.6.28-10	4	508	176	180	220	224	168	248	256	228	236	✓
2.6.30-1	4	496	220	224	268	272	192	296	304	276	284	✓
2.6.31-1*	4	500	220	224	268	272	192	296	304	276	284	✓
2.6.32-17	4	504	228	232	268	272	200	296	304	276	284	✓
2.6.34-2	4	512	220	224	276	280	192	304	312	284	292	✓

Table 5. Detailed field offsets of `task_struct` for kernel version inference

data structure `task_struct` satisfies the above requirements for most of the kernels. The offsets and types of fields in `task_struct` involved in the signatures are presented from the 2nd to 12th column in Table 5. We can see that there are only two kernels (2.6.14-7 and 2.6.15-1) that cannot be distinguished using `task_struct`'s signatures. To validate, we take snapshots of these kernels and then scan the snapshots using the 13 distinct signatures. We succeed in uniquely identifying 12 of the 14 kernels. The 2 kernels that we cannot tell apart are two *consecutive* Linux kernels with no significant differences in data structure definitions.

9 Discussion

While SigGraph-based signatures are capable of identifying kernel data structure instances as demonstrated in Sections 7 and 8, we believe that there may be more sophisticated attempts to evade SigGraph in the future. In this section, we discuss possible attacks against SigGraph, assuming that the attacker has knowledge about SigGraph and has gained control of the kernel.

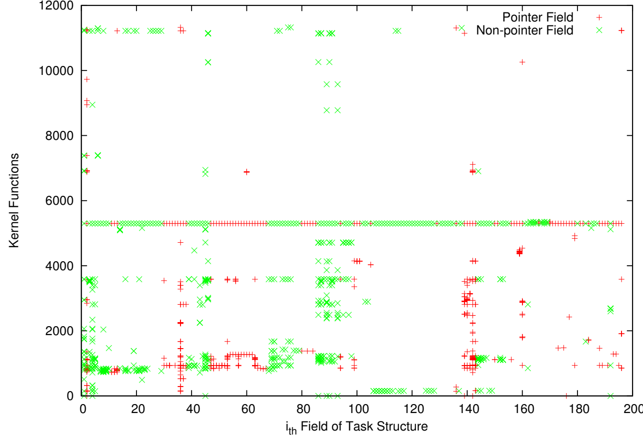
Malicious Pointer Value Manipulation. The first type of attacks are to manipulate pointers as SigGraph relies on inter-data structure topology induced by pointers. However, compared to non-pointer values, pointers are more sensitive to mutation as changes to a pointer value may very likely lead to kernel 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 attacker may try harder by destroying a pointer field after a reference, and then restoring it before its next reference. As such, it is likely that a memory snapshot not have the true pointer value. However, carrying out such attacks is challenging as there may be many code sites in the kernel that access the pointer field. All such sites need to be patched in order to respect the original semantics of the kernel, which would require a complex and expensive static analysis on the kernel. To get an (under)-estimate of the required efforts, we conduct 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 9(a), We observe that most fields are accessed by at least 6 functions. Some fields are accessed by 70 functions (the statistics is shown in Figure 9(b)). Note that these are only dynamic profiling numbers, the static counterparts may be even higher. Even if the attacker achieves some success, SigGraph can still leverage its multiple signature capability to avoid using those pointers that are easily manipulatable.

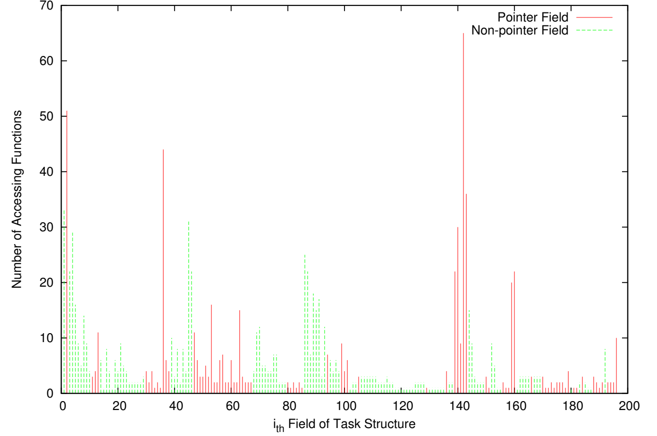
Malicious Non-Pointer Value Manipulation. Another possible way to confuse SigGraph is to mutate a non-pointer value to resemble that of a pointer. SigGraph has built-in protection against such attacks. First of all, the dynamic refinement phase will get rid of most fields that are vulnerable to such 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 [13], to study how many fields allow such cross-domain value mutation. In fact, we can effectively *integrate* SigGraph signatures with the value-invariant signatures (e.g., those derived by [13]) for the same data structure, which is likely to achieve even stronger robustness against malicious non-pointer manipulation.

Other Possible Attacks. The attacker may change data structure layout to evade SigGraph. Without knowledge about the new layout, SigGraph will fail. However, such attacks are challenging. The attacker needs to intercept the corresponding kernel object allocations and de-allocations to change layout at runtime. Furthermore, all accesses to the affected fields need to be patched.

The attacker could also try to generate *fake* data structure instances to thwart the use of SigGraph. However, we



(a) Detailed field access functions



(b) Statistics of field access functions

Figure 9. Profiling accesses to the fields of `task_struct`

point out that fake data structure instance creation is a generally hard problem across all signature-based approaches, including the value invariant-based 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 properly set up.

SigGraph can help detect kernel rootkit attacks by identifying hidden kernel data structure instances in a given memory image. There are other types of kernel attacks that do not involve data hiding (e.g., BluePill [19]). SigGraph, as a kernel object scanner generator, is not applicable to the detection of such kernel attacks.

10 Related Work

Kernel Memory Mapping and Analysis: There have been efforts in developing kernel memory mapping and data analysis techniques for kernel integrity checking (e.g., [26, 15]). Recent advances include the mapping and analysis of kernel memory images for control flow integrity checking [29] and kernel data integrity checking [27, 10]. To facilitate kernel data integrity checking, techniques have been proposed for deriving kernel data structure invariants [8, 13].

SigGraph is inspired by, and hence closely related to the above efforts [27, 8, 10, 13]. In particular, Petroni et al. [27] 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 kernel rootkits. The key observation is that any violations of semantic invariants indicate kernel rootkit presence. But the semantic invariants are manually specified. Baliga et al. [8] proposed using the dynamic invariant detector Daikon [14] to extract kernel data structure constraints. The invariants detected include membership, non-zero, bounds, length, and subset relations. Dolan-Gavitt et al. [13] proposed a scheme for generating robust value invariant-based kernel data structure signatures. Complementing these efforts, SigGraph leverages the points-to relations between kernel data structures for signature generation. As suggested in Section 9, SigGraph-based and value invariant-based signatures can be *integrated* to further improve brute force scanning accuracy.

Carbone et al. proposed KOP [10] which involves building a global points-to graph for kernel memory mapping and kernel integrity checking. The global graph is constructed via an advanced inter-procedural points-to analysis on OS source code. A few heuristics were proposed to better resolve function pointers. KOP is a highly effective system when the kernel source code and a powerful static analysis infrastructure are available. The main differences between SigGraph and KOP are the following: (1) Unlike KOP, SigGraph does not require complex points-to analysis (which often involves source code analysis) and instead only requires kernel data structure definitions. (2) KOP requires that data structure instances be reachable starting from the root(s) of the global points-to graph; whereas SigGraph does not require such global reachability and hence supports brute force memory scanning that can start at any kernel memory address. In particular, SigGraph may recognize kernel objects that are unreachable from global/stack variables. (3) To achieve robustness against pointer corruption, the global points-to graph heavily depends on a complete revelation of points-to relations between data structures; whereas SigGraph can generate multiple signatures for each data structure by *excluding* problematic pointers (e.g., `null` and `void*` pointers).

Memory Forensics: Memory forensics is a process of analyzing a memory image to interpret the state of the system.

It has been evolving from basic techniques such as string matching to more complex methods such as object traversal (e.g., [28, 34, 11, 22, 10]) and signature-based scanning (e.g., [38, 35, 13, 9, 4]). Signature-based scanning involves directly parsing the memory image using signatures. In particular, Schuster [35] presented PTfinder for linearly searching Windows memory images to discover process and thread structures, using manually created signatures. Similar to PTfinder, GREPEXEC [4], Volatility [38], and Memparser [9] are related systems capable of searching for more types of objects. Dolan-Gavitt et al. [13] further proposed an automated technique to derive robust data structure signatures. Sharing the same goal of providing robust signatures for brute force memory scanning, SigGraph provides graph-based, provably non-isomorphic signatures (as well as the corresponding memory scanners) for individual kernel data structures.

Malware Signature Derivation based on Data Structure Pattern: Data structures are one of the important and intrinsic properties of a program. Recent advances have demonstrated that data structure patterns can be used as a program's signature. In particular, Laika [12] shows a way of inferring the layout of data structure from snapshot, and uses the layout as signature. Their inference is based on unsupervised Bayesian learning and they assume no prior knowledge about program data structures. Laika and SigGraph are substantially different in that: (1) Laika focuses on deriving a program's signature from data structure patterns; whereas SigGraph focuses on deriving data structures' signatures from the points-to relations among them. (2) Laika, by its nature, does not assume availability of data structure definitions. On the contrary, data structure definitions are the input of SigGraph to generate data structure signatures.

Data Structure Type Inference: There is a large body of research in program data structure type inference, such as object oriented type inference [24], aggregate structure identification [31], binary static analysis-based type inference [6, 7, 32], abstract type inference [23, 17], and dynamic heap type inference [30]. Most of these techniques are static, aiming to infer types of unknown objects in a program. SigGraph is more relevant to dynamic techniques.

Dynamic heap type inference by Polishchuk et al. [30] focuses on typing heap objects in memory. SigGraph and [30] do share some common insights such as leveraging pointers. However, the latter focuses on type-inference of heap objects (for debugging) by assuming known start addresses and sizes of all allocated heap blocks; whereas SigGraph aims at uncovering all kernel objects (including heap, stack, and global) from a raw memory image. To uncover those objects, the user can simply execute the data structure-specific scanners on the raw memory image – without any runtime support; whereas techniques in [30] require collecting runtime information. Moreover, the different purpose of SigGraph raises the new challenge of avoiding structural isomorphism among data structure signatures.

11 Conclusion

OS kernels are rich in data structures with points-to relations between each other. We have presented SigGraph, a framework that systematically generates graph-based, non-isomorphic data structure signatures for brute force scanning of kernel memory images. Each signature is a graph rooted at the subject data structure with edges reflecting the points-to relations with other data structures. SigGraph-based signatures complement value invariant-based signatures for more accurate recognition of kernel data structures with pointer fields. Moreover, SigGraph differs from global memory mapping-based approaches that have to start from global variables and require reachability to all data structure instances from them. Our experiments with a wide range of Linux kernels show that SigGraph-based signatures achieve zero false negative rate and very low false positive rate. Moreover, the signatures are not affected by the absence of global points-to graphs and are robust against pointer value anomalies and corruptions. We demonstrate that SigGraph can be applied to kernel memory forensics, kernel rootkit detection, and kernel version inference.

Acknowledgement

We would like to thank our shepherd, Phil Porras, and the anonymous reviewers For their insightful comments and suggestions. Special thanks go to Michael Locasto, Weidong Cui and AAron Walters for their constructive feedback that has helped improve the paper. This research was supported, in part, by the National Science Foundation (NSF) under grants 0716444, 0720516, 0845870, 0852131, 0855036, and 1049303. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

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] O. Arkin, F. Yarochkin, and M. Kydyraliev. "the present and future of xprobe2: The next generati on of active operating system fingerprinting. sys-security group,, July 2003.

[6] G. Balakrishnan, , G. Balakrishnan, and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'04)*, pages 5–23. Springer-Verlag, 2004.

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

[8] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*, pages 77–86, Anaheim, California, December 2008.

[9] C. Betz. Memparser. http://sourceforge.net/projects/memparser.

[10] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 555–565, Chicago, IL, USA, 2009.

[11] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *Digital Investigation*, 5(Supplement 1):S65 – S75, 2008. The Proceedings of the Eighth Annual DFRWS Conference.

[12] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, pages 231–244, San Diego, CA, December, 2008.

[13] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM.

[14] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering*, 27(2):1–25, 2001.

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

[16] L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–10. USENIX Association, 2007.

[17] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA'06)*, pages 255–265, Portland, Maine, USA, 2006. ACM.

[18] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, pages 128–138, Alexandria, Virginia, USA, 2007. ACM.

[19] B. Laurie and A. Singer. Choose the red pill and the blue pill: a position paper. In *Proceedings of the 2008 workshop on New security paradigms*, pages 127–133, 2008.

[20] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

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

[22] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*, pages 23–32, Anaheim, CA, 2005. USENIX Association.

[23] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th international conference on Software engineering*, pages 338–348, Boston, Massachusetts, USA, 1997. ACM.

[24] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161, Phoenix, Arizona, United States, 1991. ACM.

[25] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[26] 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*, pages 179–194, San Diego, CA, August 2004.

[27] 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*, Vancouver, B.C., Canada, August 2006. USENIX Association.

[28] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[29] 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'07)*, pages 103–115, Alexandria, Virginia, USA, October 2007. ACM.

[30] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not.*, 42(1):39–46, 2007.

[31] G. Ramalingam, J. Field, and F. Tip. 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)*, pages 119–132, San Antonio, Texas, 1999. ACM.

[32] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'08)*, pages 16–35, 2008.

[33] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection*, Ottawa, Canada, September 2010.

[34] J. Rutkowska. Klister v0.3. https://www.rootkit.com/newsread.php?newsid=51.

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

[36] M. Smart, G. R. Malan, and F. Jahanian. Defeating tcp/ip stack fingerprinting. In *Proceedings of the 9th conference on USENIX Security Symposium*, pages 17–17. USENIX Association, 2000.

[37] I. Sutherland, J. Evans, T. Tryfonas, and A. Blyth. Acquiring volatile operating system data tools and techniques. *SIGOPS Operating System Review*, 42(3):65–73, 2008.

[38] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.