Graph-based Signatures for Kernel Data Structures

Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu
Purdue University
{zlin,rhee,xyzhang,dxu}@cs.purdue.edu

Xuxian Jiang
NCSU
jiang@cs.ncsu.edu

February 6, 2010

Abstract

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

1 Introduction

Given a kernel data structure definition, identifying instances of that data structure in a kernel memory image is a highly desirable capability in many computer security and forensics applications, such as memory forensics [24, 9, 18, 32, 30], kernel data integrity check and rootkit detection [22, 8, 11, 23, 6], and virtual machine introspection [15, 21]. Analogous to the pattern recognition problem in image processing, the problem of data structure (especially kernel data structure) instance recognition has received increasing research attention. For example, the state-of-the-art solutions often rely on the *field value invariance* exhibited by a data structure as its signature [33, 31, 11, 7, 6]. The value of such a field is either constant or in a fixed range and the effectiveness and robustness of the value-invariant approach has been well demonstrated. Yet there exist many kernel data structures that are not covered by the value-invariant signatures. For example, there are data structures that do not have fields with invariant values or value ranges. It is also possible that an invariant-value field is corrupted (e.g., by kernel bugs or attacks), making the corresponding data structure instance un-recognizable. Furthermore, some value-invariant based signatures may not be unique enough to distinguish themselves from others, e.g., a signature that demands the first field to be 0 can easily generate a lot of false positives.

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

```
struct task_struct {
  [0] struct thread_info *thread_info;
  [4] struct mm_struct *mm;
  [8] struct linux_binfmt *binfmt;
 [12] struct task_struct *parent;
}
```

(a)

```
struct thread_info {
  [0] struct task_struct *task;
}
```

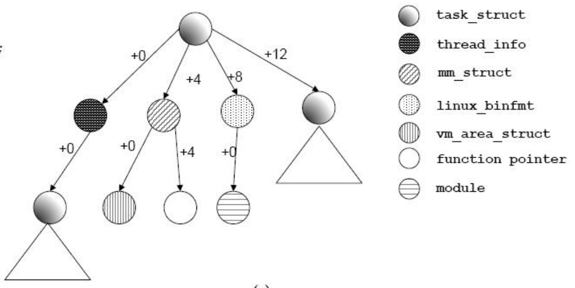
(b)

```
struct memory {
  [0] struct vm_area_struct *mmap;
  [4] void (*map_area)(struct memory*
      mmap, long unsigned int);
}
```

(c)

```
struct linux_binfmt {
  [0] struct module *module;
}
```

(d)



(e)

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

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

The graph-based signature has the following key properties: (1) It models the topological invariants between the subject data structure and those directly or indirectly reachable via points-to relations, which complements the existing value invariant based approach. (2) For a data structure with pointer fields, it is likely that the (recursively defined) signature graph is unique to that data structure. (3) Unlike other approaches (e.g.,[8]), because of the large existence of pointer fields, SigGraph can avoid the complex and expensive points-to analysis (for `void` pointers) by simply discarding the destination type checking as there exist a wealthy type-clear pointers. (4) The graph pattern can often be described by *context-free-grammars* such that parsers/scanners can be automatically generated to recognize data structure instances. To determine if address x holds an instance of data structure T, we only need to perform pattern matching starting at x. This avoids the construction of a *global* memory graph starting from the global variables and stack variables of a program [17, 8] and hence tends to be more independent and robust. (5) Effective algorithms can be devised to determine the existence of unique graph signatures and also generate such signatures.

Based on the above properties, SigGraph automatically extracts graph signatures of kernel data structures with pointers. For each graph signature, SigGraph further generates a parser routine for performing pattern recognition on a kernel memory image using that signature. To deal with `null` pointer and other practical issues at runtime, we refine the signatures through dynamic profiling and training. We have performed extensive evaluation on the graph-based signatures on several Linux kernels and verified the signatures' uniqueness. Our signatures achieve low false positives and zero false negatives when applied to data structure instance recognition on kernel memory images. Finally, our experiments show that SigGraph is robust in the face of field value (including pointer field) corruption and in the absence of global memory maps.

2 Overview

2.1 Problem Statement and Challenges

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

At runtime, if the pointer fields do not have a `null` value, they should point to their corresponding target types. Let $S_T(x)$ denote a boolean function that decides if the memory region starts at x is an instance of type T and $*x$ denote the address pointed to by the value in x. Take `task_struct` structure as an example, we have the following rule, assuming all pointers are not `null`.

$$S_{\texttt{task\_struct}}(x) \;\rightarrow\; S_{\texttt{thread\_info}}(*(x+0)) \wedge S_{\texttt{memory\_struct}}(*(x+4)) \wedge S_{\texttt{linux\_binfmt}}(*(x+8)) \wedge S_{\texttt{task\_struct}}(*(x+12)) \tag{1}$$

It means that if $S_{\texttt{task\_struct}}(x)$ is true, 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

$$S_{\texttt{thread\_info}}(x) \;\rightarrow\; S_{\texttt{task\_struct}}(*(x+0)) \tag{2}$$

$$S_{\texttt{mm\_struct}}(x) \;\rightarrow\; S_{\texttt{vm\_area\_struct}}(*(x+0)) \wedge S_{\texttt{map\_area\_fun\_t}}(*(x+4)) \tag{3}$$

$$S_{\texttt{linux\_binfmt}}(x) \;\rightarrow\; S_{\texttt{module}}(*(x+0)) \tag{4}$$

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

$$S_{\texttt{task\_struct}}(x) \;\rightarrow\; S_{\texttt{task\_struct}}(*(*(x+0)+0)) \wedge S_{\texttt{vm\_area\_struct}}(*(*(x+4)+0)) \; S_{\texttt{map\_area\_fun\_t}}(*(*(x+4)+4)) \wedge S_{\texttt{module}}(*(*(x+8)+0))) \wedge S_{\texttt{task\_struct}}(*(*(x+12)+0)) \tag{5}$$

The rule describes a structural pattern as shown in Figure 1 (e), where nodes represent pointer fields with patterns denoting pointer types, edges represent the points-to relations, and the triangles represent the recursive occurrences of the same pattern. This means that if the memory region starting at x is an instance of `task` structure, the region must assume the given structural pattern. Note that the inference in rule 5 is one directional (from left to right), we observe that *the structural patterns of many data structures are so unique that the reverse inference is likely true*. In other words, we can use the structural pattern (the right hand side of the rule) as the signature of the data structure and achieve the reverse inference as follows (observe the direction of the inference has changed).

$$S_{\texttt{task\_struct}}(x) \;\leftarrow\; S_{\texttt{task\_struct}}(*(*(x+0)+0)) \wedge S_{\texttt{vm\_area\_struct}}(*(*(x+4)+0)) \; \ldots \tag{6}$$

At the high level, SigGraph works as follows. It takes kernel data structure definitions and generates the graph-based structural signatures for each data structure if there exists a unique signature. Parsers are further generated from the signatures so that they can be used to scan a given memory image to identify instances of the corresponding data structures.

We need to address a number of challenges in developing SigGraph:

- **Deciding uniqueness.** Given static data structure definitions, we aim to construct their structural patterns based on the points-to relations. However, it is possible that two distinct data structures may have isomorphic structural patterns such that there exists no pattern that can distinguish instances of the two data structures. Hence, our first research challenge is to identify the sufficient and necessary conditions for signature uniqueness.

- **Generating unique signatures.** It is possible that a data structure may have multiple unique signatures, depending on the way we expand the pointer fields. Finding the set of minimal signatures that have the smallest size and retains uniqueness is a combinatorial optimization problem. Our second challenge is to design an algorithm for signature generation.

- **Handling `null` pointers.** Although statically a data structure may have a unique structural pattern, at runtime, as pointers may be null; on the other hand, non-pointer fields may have pointer-like values, part of the pattern may not manifest. We need to handle these cases and generate robust signatures.

- **Automatic parser generation.** In order to achieve generality, our technique shall be able to automatically generate a parser from the signature description, which is also automatically generated from the data structure definition. A parser is a stand alone program that scans a given memory snapshot and reports the instances for the given data structure.

We will discuss in details how we address these challenges later in the paper.

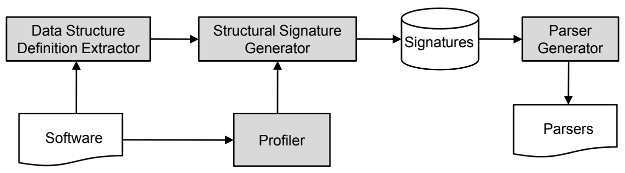

Data Structure Definition Extractor
Structural Signature Generator
Signatures
Parser Generator
Software
Profiler
Parsers

Figure 2: SigGraph system overview

2.2 System Overview

The overview of our SigGraph system is shown in Figure 2. It consists of four key components: (1) data structure definition extractor, (2) dynamic profiler, (3) signature generator, and (4) parser generator. The starting point of our system is to extract data structure definitions for the target software. In this paper, we use Linux kernels as the subject programs. SigGraph extracts all kernel data structure definitions automatically through a compiler pass. Because our signatures rely on pointer fields and pointers may be `null` or some special value because of the specific kernel programming practice or points-to user space,the dynamic *profiler* identifies such problematic pointer fields and handle them separately. The *signature generator* is responsible for checking if unique signatures exist for a data structure because multiple data structures may have isomorphic structures that make them indistinguishable. If there are unique signatures, the generator is also responsible for generating one of them. The generated signatures are passed to the *parser generator* component that can automatically generate parsers for individual data structures from their signatures. Such parsers can be used to scan memory to identify data structure instances. Note that, as the end product of our technique, the parsers can be shipped independently to users for different applications, such as kernel memory forensics and kernel rootkit detection.

3 Data Structure Definition Extraction

A naive method to extract data structure definitions is to directly extract them by processing source code plain text. This method is error-prone because definitions may be present in multiple source files; they may have various scopes (i.e., declaration contexts) and type aliases can be introduced by `typedef`. Our current prototype uses a compiler based approach. We instrument the compiler to walk through source code and extract data structure definitions. It is robust as it is based on a full-fledged language frontend. In particular, we introduce a compiler pass in `gcc-4.2.4`. The pass takes abstract syntax trees (ASTs) as input because AST retains substantial symbolic information for data structures, such as types, scopes, field offsets and sizes, memory alignments, file names and locations [1]. Using a compiler pass also allows us to easily handle data structure in-lining. Data structure in-lining occurs when a 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 recognize pointer types through ASTs, which is hard to achieve otherwise. Note that in large scale programs type aliases for pointers are very common, e.g., it is a regular coding style to assign an alias to a pointer type through `typedef`.

The net outcome of the compiler pass is data structure definitions extracted in a canonical form, which will be used by other components. The pass is inserted in 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 are dumped to a file, which is indexed by a tuple of `<name, file>` with `name` being the symbolic name of the data structure and `file` being the declaration file. The reason for using a pair to index a data structure is that in large scale programs, it is possible two different data structures in two respective files have the identical name. For example, in Linux kernel 2.6.18-1, we find that more than three hundred data structures (roughly 3%) have name conflict with others. In order to precisely reflect the field layout after data structure in-lining. We flatten nested definitions and adjust offsets. A slight worth to note issue is if a data structure does not have a name, we will assign an internal name and make that data structure only visible to that particular compiling file.

```
struct A {                    struct X {
  [0]  struct B * a1;           ...
   ...                          [8]  struct Y  * x1;
  [12] struct C * a2;            ...
   ...                          [36] struct BB * x2;
  [18] struct D * a3;            ...
}                               [48] struct CC * x3;
                                 ...
                                [54] struct DD * x4;
                              }

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

Figure 3: Insufficiency of pointer layout uniqueness.

Note that using source code is *not* a fundamental requirement of our technique but rather an implementation decision. If debug information is provided with the binary, SigGraph can simply rely on the debug information, which usually contains detailed definitions of all the data structures used by the program.

4 Signature Generation

Data structure definitions often give rise to structural/topological properties between data structures based on points-to relations. Given a data structure T, assume it has n pointer fields with offsets f_1, f_2, ..., f_n and types t_1, t_2, ..., t_n. A predicate $S_t(x)$ determines if the region starts at the address x is an instance of t. The following production rule can be generated for T.

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

It means that if the region starts at x is an instance of T, then it must be true that the regions pointed to by the pointer fields have the corresponding types. Sample rules can be found in rules (1)-(4) in Section 2.

While the above rule describes that if we know a given region is an instance of a data structure, then we know that the structural properties must be satisfied, however, our research problem is along the reverse direction. Given a memory snapshot, we want to identify instances of data structures by looking for satisfactions of the structural properties. In particular, it is unlikely for us to know the types of individual memory locations based on their values. However, we can decide if a memory location contains a pointer and hence we know the layout of pointers. We can also traverse along points-to edges and look for the pointer structure of the lower layers. Therefore, the core challenge is to *find a unique graph-based structural representation with only pointers and pointer field offsets for a target data structure.* We use such a graph as the signature of the target data structure (at the root of the graph). For simplicity of discussion, we assume pointers are not null in this section and a pointer has one explicit type (not a `void *` pointer). We will discuss how to handle those practical issues in Section 6.

If two data structures have the same pointer field layout, we need to further look into the structures of the regions reached through the points-to edges. We also call such regions the lower layers. We observe that *even though the pointer field layout of a data structure may be unique (different from any other data structures), an instance of such layout in memory is not necessary an instance of the data structure.* Consider Figure 3, `struct A` and `X` have different layouts for their pointer fields. If the program has only these two data structures, it appears that we can use their one level pointer structure as the signature. However, this is not true. Consider the memory segment on the bottom of Figure 3, in which we detect three pointers (the boxed bytes). It appears that $S_A(0xc80b20f0)$ is true because it fits the one level structure of `struct A`. But it is also possible that the three pointers are the instances of fields `x2`, `x3`, and `x4` in `struct X` and hence the region is part of an instance of `struct X`. The reason is that the one level structure of `A` coincides with the sub-structure of `X`.

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

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

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

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

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

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

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

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

The strings [1] and [2] are both pointer patterns of A. The pointer patterns of a data structure are candidates for its signature. As one data structure may have many pointer patterns, the challenge becomes to algorithmically select the minimal unique pointer pattern of a given data structure so that instances of the data structure can be identified from memory by looking for satisfactions of the pattern.

Existence of Signature. The first question we need to answer is whether a unique pointer pattern exits 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 structure of the lower layers to distinguish it from the other data structures. However, it is possible that T is not distinguishable from another data structure R if their structures are isomorphic.

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

T and R are isomorphic, denoted as $T \bowtie R$, if and only if
[1] $n \equiv m$;
[2] $\forall 1 \le i \le n \ \boxed{f_i \equiv g_i}^{[2.1]} \wedge (\boxed{t_i \bowtie r_i}^{[2.2]} \vee \boxed{\text{a cycle is formed when deciding } t_i \bowtie r_i}^{[2.3]})$;

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

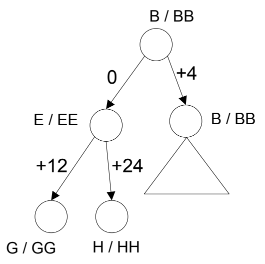
Figure 4 (a) presents the definitions for some data structures in Figure 3. The data structures whose definitions are missing from the two figures do not have pointer fields. According to Definition 3, $B \bowtie BB$ because they both have two pointers at the same offsets; and the types of the pointer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

Given a data structure, now we can decide if it has a unique signature. As mentioned earlier, we assume an ideal environment in this section. Pointers are not null and do have have a `void*` type. Non-pointer fields do not assume a pointer value.

```
struct B {              struct BB {              struct D {
  [0]  E * b1;            [0]  EE * bb1;            ...
  [4]  B * b2;            [4]  BB * bb2;          [4]  I * d1;
}                       }                        }

struct E {              struct EE {              struct DD {
   ...                     ...                      ...
  [12] G * e1;            [12] GG * ee1;          [8]  II * dd1;
   ...                     ...                   }
  [24] H * e3;            [24] HH * ee3;
}                       }
```

(a) definitions



(b) structures of B and BB

Figure 4: Data structure isomorphism.

Theorem 1. *Given a data structure T, if there does not exist a data structure R such that*
[1] *$IPP(T)$ is a sub-pattern of $IPP(R)$, and*
[2] *assume the sub-pattern in $IPP(R)$ is $g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$, $t_1 \bowtie r_1$, $t_2 \bowtie r_2$, ... and $t_n \bowtie r_n$,*

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

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

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

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

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

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

$\square$

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

Note that two isomorphic data structures may have different concrete pointer field types. But given a memory snapshot, it is unlikely for us to know the concrete types of memory cells. Hence these pointer types serve as no-ops in the pointer patterns. Their presence is rather for readability.

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

It is worth noting that the theorem only specifies the conditions to distinguish T with any other individual data structure R. It is possible in a real memory snapshot, multiple data structures are allocated next to each other and hence they constitute a structural pattern isomorphic to T's. However, we haven't found such cases in practice.

Finding the Minimal Signature. Even though we can decide if a data structure T has a unique signature with the theorem, there may be multiple pointer patterns of T that can distinguish it from other data struc-

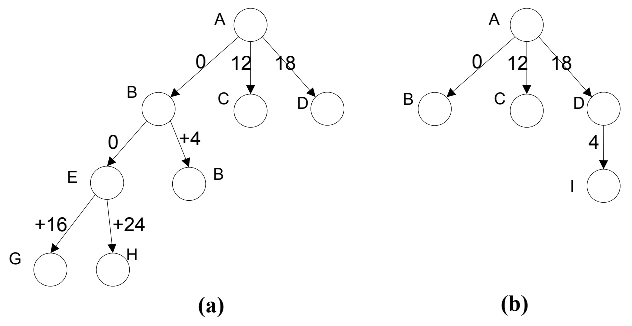


Figure 5: If the offset of field `e1` in `struct G` changes to 16, `struct A` has two possible signatures.

tures. Ideally, we want to find the minimal pattern as it requires the minimal parsing efforts during scanning. For example, if the field `e1` in `struct G` changes to 16, `struct A` has two possible structures as shown in Figure 5. They correspond to the pointer patterns

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

and

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

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

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

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

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

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

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

5 Parser Generation

Given a data structure signature, i.e., a pointer pattern, our technique can automatically generate a parser. The parser scans memory and reports satisfactions of the pattern, which are instances of the data structure. In order to automatically generate parsers, we describe all signatures using a context free grammar (CFG). Then we leverage `yacc` to generate parsers. The CFG is described as follows.

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

In the above grammar, **number** and **type** are terminals that represent numbers and type symbols, respectively. A $Signature$ is a sequence of **number** $\cdot$ $Pointer$, in which $Pointer$ describes either the **type** or the

Algorithm 1 An approximate algorithm for our signature generation

Input: Data structure T;

Output: The pointer pattern that serves as the signature.

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

```
int isInstanceOf_A(void *x) {
  x=x+0;
  {
     y=*x;
     y=y+0'
     assertPointer(*y);
     y=y+4;
     assertPointer(*y);
  }
  x=x+12;
  assertPointer(*x);
  x=x+6;
  {
     y=*x;
     y=y+4;
     assertPointer(*y);
  }
  return 1;
}
```

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

$Signature$ of the data structure being pointed-to. It is easy to tell that the grammar describes all the pointer patterns in Section 4, including the signature of A generated by our technique (Equation (9)).

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

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

For example, assume a data structure T has a unique signature $0 \cdot A \cdot 8 \cdot B \cdot 12 \cdot C$. If there is a pointer array that stores a consecutive sequence of pointers, even though the signature is unique and has no structural conflict with any other data structures, the generated parser will mistakenly identify part of the array as an instance of T.

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

```
int isInstanceOf_T(void *x) {
  x=x+0;
  assertPointer(*x);    // field of type "A *"
  x=x+4;
  assertNotPointer(*x); // field of non-pointer
  x=x+4;
  assertPointer(*x);    // field of type "B *"
  x=x+4;
  assertPointer(*x);    // field of type "C *"
}
```

Note for the memory deference $*x$ operation, when applying in live memory analysis, there is no problem to find out their corresponding memory address. If it is used in off line analysis, we need to do virtual to physical address translation. Details on how we perform the off line analysis is discussed in the evaluation section.

6 Handling Practical Issues

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

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

2. **Void Pointer –** Some of the pointer fields have a `void*` type, creating troubles for our signature generation algorithm because at runtime, the pointer may have variable types, suggesting variable structures.

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

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

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

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

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

We find that most of the above problems are essentially caused by the confusion when deciding pointer or non-pointer fields. Fortunately, the following observation leads to a simple solution: Pruning a few noisy pointer fields will 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 unstable pointer/non-pointer fields.

Our profiler relies on a virtual machine monitor QEMU [3] to keep track of kernel memory allocation and deallocation for kernel data structures. More specifically, since most kernel objects are managed by slab allocators, we hook the allocation and deallocation of `kmem_cache` objects through functions such as `kmem_cache_alloc` and `kmem_cache_zalloc`, retrieving the function arguments and return values to track these objects. Their types are acquired by looking at their slab name tags – for example, if a slab object is pointing to the name of "`task_struct`", we know that object is an instance of `task_struct`. Then we track the life time of these objects, and monitor their values. More details on how we track the allocation/deallocation of kernel objects at VMM level can be found in our technical report [29].

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

With the above profiles, we conduct the following refinement: (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 not considered pointer fields. We annotate them with a special symbol to indicate that they should *not* be considered as non-pointer fields either, so that method `assertNonPointer()` discussed in Section 5 will not be applied to such fields; (3) excluding all the pointer fields that have ever had a `null` value or a non-pointer value during profiling; as well as all non-pointer fields that ever have a pointer value during profiling. Neither `assertPointer()` nor `assertNonPointer()` will be applied to these fields; (4) allowing pointers to have special value 0x00100100 or 0x00200200.

7 Evaluation

We have implemented a prototype of SigGraph, with C and Python code. Specifically, we instrument `gcc-4.2.1` to traverse the ASTs and collect the data structure definitions, and we implement our signature generator in python and C. Our parser generator is lex/yacc based, and the generated parsers are in C. The total implementation is around 9.5K lines of C code and 6.8K LOC python code.

7.1 Uniqueness of the Signature

We first test if unique signatures exist for kernel data structures. We take 5 popular Linux distributions, which are shown in the first column of Table 1. Then we compile the corresponding kernels using the 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% of the data structures with pointer fields have unique signatures. We show the average step of expansions needed to generate unique signatures (column `Average depth`). We also show the number of unique signatures at various steps of expansions in Table 2. 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 levels signatures. Because of subgraph isomorphism, there are data structures that do not have any unique signatures. The total number of such structures is shown in the last column. Note that these are all static numbers before the dynamic refinement.

Distribution	Kernel version	#Total structs	#Pointer structs	Percentage	Signature Statistics			
					#Unique signature	Percentage	Average depth	#No signature
Fedora Core 5	2.6.15-1	8850	3597	40.64%	3229	89.76%	2.31	368
Fedora Core 6	2.6.18-1	11800	4882	41.37%	4305	88.18%	2.45	572
Ubuntu-7.04	2.6.20-15	14992	6096	40.67%	5395	88.50%	2.54	701
Ubuntu-8.04	2.6.24-26	15901	6427	40.42%	5645	87.83%	2.47	782
Ubuntu-9.10	2.6.31-1	26799	9957	37.15%	8683	87.20%	2.73	1274

Table 1: Experimental result on our signature uniqueness testing

Kernel version	Number of signatures in different steps												
	1	2	3	4	5	6	7	8	9	10	11	12	13
2.6.15-1	1355	823	461	229	76	194	85	4	1	0	1	-	-
2.6.18-1	1820	1057	382	410	159	337	121	9	3	5	1	1	-
2.6.20-15	2137	1311	680	236	407	501	106	9	1	5	1	1	-
2.6.24-26	2172	1316	761	475	624	248	37	7	1	0	3	1	-
2.6.31-1	3364	1951	696	319	1492	494	344	19	1	0	1	1	1

Table 2: Detail statistics on our static signatures

7.2 Effectiveness of the Signature

To test the effectiveness of our system, we take Linux kernel 2.6.18-1 as a working system, and show how the generated signatures can detect data structure instances. We list 23 widely used kernel data structures which are shown in the 2^{nd} column of Table 3. We choose these data structures because: (1) they are the most commonly examined data structures in existing literature [24, 9, 18, 32, 30, 33, 31, 7]; (2) they are very important data structures that can show the status of the system in the aspects of process, memory, network and file system; from these data structures, we can reach most of the kernel objects; and (3) they contain pointer fields. Note that when parsing instances for these data structures, other data structures are also traversed, as our signatures often contain lower level data structures.

The size of the data structures is shown in the 3^{rd} column of Table 3. To ease the presentation, 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 4^{th} and 5^{th} columns, respectively.

7.2.1 Experiment Setup

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

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

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

Virtual to Physical Address Translation Because we use graph structure as signature, and we need to traverse pointers, we need to resolve the destination addresses of the pointers in the physical memory. That is, we need to perform virtual address to physical address translation as pointer values are all on virtual space. To this end, we need the provisional page global directory, `swapper_pg_dir`. There is a very strong signature for page directories (i.e., the `pgd` object). We scan the physical RAM by looking for the identical 63 consecutive 4 bytes sharing the last 9 bits $1e3$. These values correspond to the direct mapping of a

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

Table 3: Summary of the signature for our testing Linux Kernel 2.6.18-1

kernel memory address range from $0xc0400000$ to $0xcfffffff$ into physical addresses using 4MB pages. Especially the earliest one matches `swapper_pg_dir`, which allows us to translate kernel addresses.

7.2.2 Dynamic Refinement

In this experiment, we carry out the dynamic refinement phase as described in Section 6. The depth and the size of signatures before and after pruning are presented in the **Our Signature** columns in Table 3, with D the depth and $\sum|P|$ the number of pointer fields. Note that the signature generation algorithm has to be run again on the pruned data structure definitions to ensure uniqueness. Observe that since pointer fields are pruned and hence the graph topology gets changed, our algorithm has to perform a few more expansions to redeem uniqueness, and hence the depth of the signature increases after pruning for some data structures, such as `task_struct`.

7.2.3 Value Invariant based Signatures

To compare our approach with value invariant based signatures [33, 31, 11, 7], we also implemented a basic value-invariant signature generation system and tested it. In particular, we generally derive four types of invariant for each fields, (1) zero-subset: check if there is a subset of the values that is zero among the true instances. If so, ignore the remained checking for this field; (2) constant: check if the value is always constant across all instances; (3) bitwise-AND: check if performing a bitwise AND of all instances, we can get a non-zero value – to capture the common bits; and (4) alignment: check if there is a power of two (other than 1) number on which all instances are well-aligned.

To derive these value invariants, we perform two types of profiling: one is access frequency profiling (to prune out the fields that are never accessed by the kernel), the other is to sample their values and produce the signatures. The access frequency profiling is achieved by instrumenting QEMU to track memory reads and writes to these fields. Sampling is similar to the sampling in our dynamic refinement phase.

All the data structures have value invariants, and the statistics of these signatures are provided in the last four column of Table 3. The total numbers of zero-subset, constant, bitwise-AND, and alignment are denoted as $|Z|$, $|C|$, $|B|$, and $|A|$, respectively.

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

Table 4: Experimental results of our graph based signature and value-invariant signature

7.2.4 Results

The final results for each signature when scanning a test image is shown in Table 4. The second column shows the total number of true instances of the data structure, which is acquired by the modified `crash` utility [2]. The $|R|$ column shows the number of instances the signature scanning detected. Due to the limitation of `crash`, these objects have to be live, i.e., reachable from global or stack variables. However, signature approaches are able to identify free objects as well. Hence, the detected free objects are determined as false positives (FPs) based on the ground truth from `crash`. We further take the free but haven't-been-destroyed object, which can also be traversed by crash if the slab allocator haven't released it to free pages, to better evaluate the real FPs. We present the FPs without considering free objects in the $|FP|$ column and the FPs considering free objects in the $|FP'|$ column. The false negative FN is computed by comparing with the ground truth objects from `crash`.

Observe that among the examined 23 data structures, when free objects are not considered, there are 16 that our approach precisely and completely identifies all instances (both FP and FN are zero). However, value invariant has only 5 such data structures. If we consider free objects, 20 can be perfectly identified with our approach, whereas 9 can be detected via the value based approach. Note for value invariant signature system, the best way to interpret the high false positive rate for some data structures is to throw away the derived signature because it has high false positive; that is, value invariant system can report no signature for these data structures. But for the 23 data structure we listed, our system do have signatures.

Also, we carefully examine the false positive cases of the 3 data structures, we find the following.

- For `vm_area_struct` case, we have 9 false positives among the total 2233 detected instances. After dynamic refinement, some pointer fields are pruned, such as the pointer field at offset 12 (as shown in Figure 7(a)). Finally, the generated unique signature contains only the first layer pointer structure, in particular, it consists of a pointer field at offset 0 (`mm_struct`), and then a sequence of non-pointer fields, and so on. However, the `task_structure` starting from offset 156 has the identical sub-pattern except the offset 160 is a pointer. But in some rare occasions (which are not captured by our profiler), the pointer field at offset 160 could be 0, leading to a false positive. This is due to the difference between the training images and the test image. We find 9 FPs in this case.

- We have 2 FPs for `dentry`, which are shown in Figure 7(b). We classify these two instances as FPs

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

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

(a) False Positive of vm_area_struct

```
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
```

(b) False Positive of dentry

```
 struct sysfs_dirent {
    [0] atomic_t s_count;
    [4] struct list_head s_sibling;
   [12] struct list_head s_children;
   [20] void *s_element;
   [24] int s_type;
   [28] umode_t s_mode;
   [32] struct dentry *s_dentry;  [pruned]
   [36] struct iattr *s_iattr;    [pruned]
   [40] atomic_t s_event; }
fp1
0xcffaeffc: 0x00000000  0xcffa3800  0xcffaf800  0xcffa3808
0xcffaf00c: 0xcffaf808  0xcffc2800  0x00000000  0x00000000
0xcffaf01c: 0xcfd9bde0  0x00000008  0x70008086

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

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

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

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

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

(c) False Positive of sysfs_dirent

Figure 7: False Positive analysis

because they cannot be found in either the pool of live objects or the pool of free objects. However, if we carefully check each field value, especially the boxed ones: the 0xdead4ead (SPINLOCK MAGIC) and 0xcf91fe00 (a pointer to `dentry_operations`), it is hard to believe these are not `dentry` instances. As such, we suspect they are not FPs, and they are the cases that the slab allocator has freed the memory page of the destroyed `dentry` instances.

- We have 6 FPs in `sysfs_dirent` data structure among the 2116 detected instances. The detailed memory dump of these 6 FP cases is shown in Figure 7(c). After our dynamic refinement, the fields at offsets 32 and 36 are pruned because they often contain `null` pointers, and the final signature entails checking two `list_head` data structures followed by a `void*` pointer at offset 4, 8, 12, 16 and 20, and four non-pointer field checking. Note one `list_head` has only two fields: previous and next pointer. However, there are 6 memory chunks that match our signature in the testing image. The chunks are not captured as part of the ground truth of any data structures. We suspect that it could be the case that they are aggregations of multiple data structures and the aggregations coincidentally manifest the pattern.

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

We also observe that signature approaches, including both our approach and value based approaches, can be used to identify free objects, such as in the cases of data structures `mm_struct` and `fs_struct`. Our approach performs slightly better and has identified more free object such as in the case of `dentry`, `inode_security`, and `mb_cache_entry`.

7.3 Robustness

In the absence of global variables and memory graph The uniqueness of SigGraph lies in the capability of locally discovering the instance without fully traversing a global memory graph. In this experiment, we develop a rootkit which tries to do something bad, and then overwrites some of the important "root" pointers to crash the kernel (as a denial of service attack demo) thus evading the memory graph-based analysis.

In particular, we craft a rootkit (a kernel module per se) to overwrite several global variables which are related to process list and the management of slab cache. Because most memory graph based approach (e.g., the `crash` utility [2]) relies on `task_struct` traversal to reach many other kernel objects, and slab cache traversal to list all slab objects. Our rootkit overwrites the global variable `pid_hash`, which points to `pid_hash` array where a list of task structures is stored. Besides `pid_hash` which has the information about the running `task_struct`, this rootkit also clears the content of `init_task`, another global variable which is the root of all processes. The third place our rootkit clears is the slab cache. A slab cache is a group of the same kernel memory objects. This memory group is itself managed by using a data structure (`kmem_cache`) and often it is pointed to by a static kernel pointer. By clearing these pointers, we can effectively unlink the valid references to slab caches that manage slab objects. As such, our rootkit also overwrites the global variable to `task_struct_cachep`.

After removing these pointers, the kernel crashed as expected. Thus, before loading our rootkit, we took a snapshot, which has 78 running process. Then we run our rootkit, and then the whole system crashed and we took another image. Next, we run the `crash` utility (a memory graph based approach system) on these two images, for the first image, `crash` reported there are 78 processes, but for the second one, unfortunately, it reported "invalid kernel virtual address: 0 type: `pid_hash` contents".

Then, we run our `task_struct` signature parser to scan the process instance, for the first one, we identified 78, and for the second one, we reported 77 because the `init_task` has all been cleared. Therefore, we believe our SigGraph signature is more robust and does not rely on accurate construction of the global memory graph.

In the face of malicious pointer manipulation To evade our signatures, an attacker has to mutate pointer values. Basically, he has three ways: (1) changing the pointer value to zero because our signature will ignore `null` pointers, (2) changing the pointer to other object with the same types (to ensure kernel will not crash), and (3) changing the pointer value to any garbage but disable kernel refer them.

In this experiment, we also take the `task_struct` as the attack target, and demonstrate the consequence of the three cases. In our signature for `task_struct`, we require that there are pointer fields at offset 4, 40, 44, etc. For the first case, our rootkit simply mutates the pointer fields at offset 4 and changed it zero. But the kernel immediately crashes.

In the second case, let's assume attacker has the knowledge that the offset is a `thread_info`, and his mutated pointer should point to a `thread_info` structure (note `thread_info` is a very important data structure, and it is the space of the kernel stack of process). In order to ease our observation of this rootkit, we just keep doing a `printk` with a "hello world" message to the screen. In our rootkit, we changed to another process's `thread_info`, and we succeeded, there is no crash, but the rootkit process is waiting. Apparently, this is not what attacker wishes. Also, even though the kernel did not crash, the signature shape is still there, and our scanner successfully detected it.

The third case is a bit more challenging. The attacker needs to disable the kernel pointer reference. This is possible. To demonstrate, we use the next and previous pointer of `list_head` as an example, and the attacker aims to mutate the pointer of `list_head` to evade our detection. He can first patch the next and previous pointer in the `list_head` the two pointer points-to, that is, eliminate his `list_head` from a list, then the kernel will not reference it. This time, the rootkit succeeded. However, when OS wants to de-reference other object through his `list_head`, if he didn't patch it back, the kernel will crash. That is, there is only a very short time window for this attack to succeed.

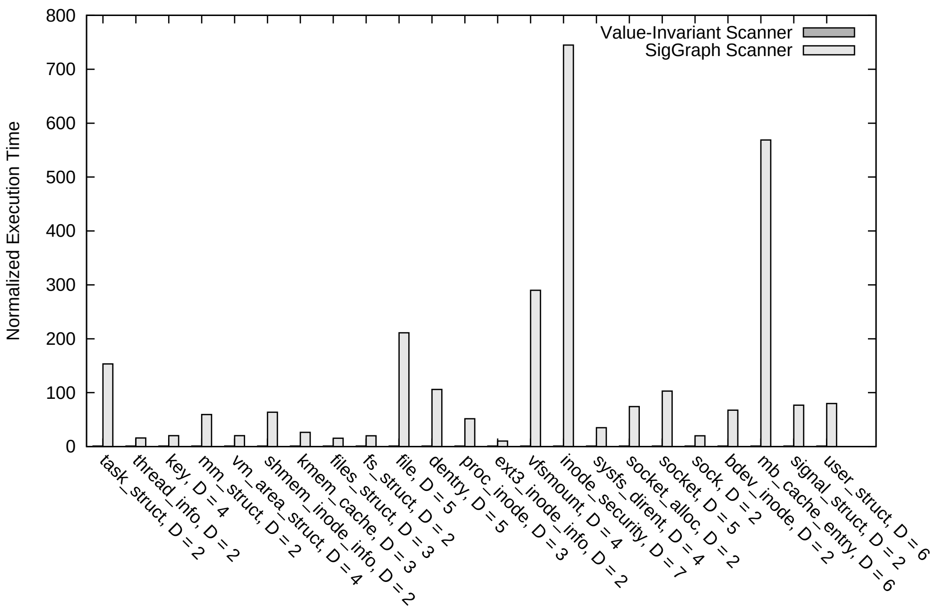


Figure 8: Performance Overhead

7.4 Performance Overhead

The last experiment we did is to test the performance overhead of our scanner (i.e., the parser). We run both our scanner and value invariant scanners on the testing image (256MB) in a machine with 3GB memory and an Intel Core 2 Quad CPU (2.4Ghz) running Ubuntu-9.04 (Linux kernel 2.6.28-17). The final result of the normalize performance overhead is shown in Figure 8. We could see the performance overhead for our scanner is still acceptable. We need to do address translation when there is a memory de-reference, but there is no need in value-invariant scanner. Thus, in all the case value-invariant inevitably performs better than our scanner. If the depth is relatively small, such as the 10 case with depth $D = 2$, our scanner only has 10X to 20X overhead than value invariant scanner. The deeper, the worse in our scanner, because more nodes need to be examined and more address translation needs to be involved, this is why the case of `inode_security` (with $D = 7$) and `mb_cache_entry` (with $D = 6$), we have big performance overhead. Thus, if the depth is not so high for the desired data structure, our system may be used as an online scanner. For example, in our experiment, it actually only takes a few seconds when scanning `fs_struct`, `thread_info`, and `files_struct`.

8 Discussion

SigGraph has a number of limitations. In this section, we examine each of them and discuss possible future directions to address them.

The first limitation is that SigGraph relies on data structure definitions. If the software does not disclose its data structure definitions, SigGraph will not work. Fortunately, for both Linux (shipped with source code) and Microsoft Windows kernel (disclosing debug symbols), SigGraph will work.

Secondly, not all data structures have pointer fields. However, value-invariant signatures will be able to handle them if their fields contain value invariants. As such, we believe a real-world memory analysis system should combine the value-invariant and graph-based signatures to achieve the maximum coverage of data structures. Also, some data structures have neither value-invariant nor graph-based signature. In such cases, we have to explore other techniques. For example, it is hard to identify the small-size data structure (e.g., the four bytes IP address) in the memory, but it is still possible to identify some instances of them if they are part of some more composite data structures that do have graph-based signatures.

Thirdly, SigGraph has a dynamic refinement component and we cannot directly use the static signature (largely because of the `null` pointer issue). Due to the nature of dynamic analysis, we cannot achieve completeness. A possible solution would be to combine the semantics of the pointer fields, and assign different weights statically, and then use them. For example, we could assign a semantic-known pointer field a heavier weight and ignore or assign a smaller weight to other pointer fields.

Finally, if there exists an un-initialized pointer and we fail to prune it off, SigGraph will have false negatives.

9 Related Work

Rootkit Detection Kernel-level rootkits pose a significant threat to the integrity of operating systems. Earlier research uses specification based approach deployed in hardware (e.g., [34, 22]), virtual machine monitor (e.g., Livewire [13]), or binary analysis [16] to detect kernel integrity violations. Recent advance includes state-based control flow integrity checking (e.g., SBCFI [25] and KOP [8]), and data structure invariant based checking (e.g., [23, 6, 11]).

Our work is inspired by the data structure invariant detection, and hence closely related to [23, 6, 11]. In particular, Petroni et al. [23] proposed examining semantic invariants (such as a process must be on either the wait queue or the run queue) of kernel data structures to detect rootkits. The key observation is that any violations of semantic invariants indicate rootkit presence. However, the extraction of semantic invariants was based on manually created rules. Afterwards, Baliga et al. [6] presented using dynamic invariant detector Daikon [12] to extract data structure constraints. The invariants detected include membership, non-zero, bounds, length, and subset relations. In contrast, we focus on structural patterns. We believe our approach is complementary to theirs. Most recently, Dolan-Gavitt at el. [11] proposed a novel system to automatically select robust signatures for kernel object signatures. Their observation is that value-invariants could be evaded by attackers and thus they propose to use fuzzing technique to test the robustness of value-invariants. The key difference is that they focus on value invariants while we focus on pointer-induced topological patterns between data structures. As discussed earlier, the best practice is likely the *integration* of the two approaches.

Malware Signature Derivation based on Data Structure Pattern Data structures are one of the important and intrinsic properties of a program. Recent advance has demonstrated that data structure patterns can be used as program signature. In particular, Laika [10] shows a way of inferring the layout of data structure from snapshot, and use the layout as signature. Their inference is based on an unsupervised Bayesian learning and they assume no prior knowledge about program data structures. Laika and SigGraph are substantially different: (1) Laika focuses on how to derive a program signature from data structure patterns, whereas SigGraph focuses on how to discover data structure instances from data structure patterns and how to derive such patterns. (2) Technically, Laika does not aim to accurately infer the data structure patterns (a large number of wrong layout could not hurt their system as they just want to have a classifier to classify the program), however we have to accurately match the instance, otherwise the high false positive and the high false negative ratios would render the system unusable. (3) Laika is not interested in any particular type of objects, whereas the output of SigGraph is the specific data structure instances.

Memory Forensics Memory forensics is a process of analyzing a memory image to explain the current state of a computer system. It has been evolving from basic techniques such as string search to more complex methods such as object traversal (e.g.,[24, 30, 9, 18, 2]) and signature based scanning (e.g., [33, 31, 11, 7]).

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

Signature scanning directly searches memory using signatures. In particular, Schuster [31] presented PTfinder for linearly searching Windows memory to discover process and thread structures, using manually created signatures. Similar to PTfinder, Volatility [33] and Memparser [7] are the other two systems that have more capabilities of searching other objects. As signatures are the key to these system, Dolan-Gavitt at el. [11] proposed an automated way to derive robust data structure signatures. SigGraph complements these systems by deriving another scheme for data structure signature generation.

Data Structure Type Inference There is a large body of research in program data structure type inference, such as object oriented type inference [20], and aggregate structure identification [27], binary static analysis based type inference [4, 5, 28], abstract type inference [19, 14], and dynamic heap type inference [26]. Most these techniques are static, aiming to infer types of unknown objects in code. SigGraph is more relevant to dynamic techniques. Dynamic heap type inference by Polishchuk et al. [26] focuses on typing heap objects in memory, using various constraints, such as size and type constraints. Heap object types are decided by resolving these constraints. The difference between their work and ours is that they assume all the heap objects are known, including their sizes and locations in the heap. This is done through instrumentation. However, SigGraph's input is simply a memory image.

10 Conclusion

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

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

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

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

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

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

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

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

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

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

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

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

[15] JIANG, X., WANG, X., AND XU, D. 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)* (Alexandria, Virginia, USA, 2007), ACM, pp. 128–138.

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

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

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

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

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

[21] PAYNE, B. D., CARBONE, M., AND LEE, W. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (December 2007).

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

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

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

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

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

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

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

[29] RHEE, J., AND XU, D. Livedm:..., 2010. Technical Report CERIAS, Purdue University.

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

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

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

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

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