Towards Automatic Inference of Kernel Object Semantics from Binary Code

Junyuan Zeng Zhiqiang Lin

The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX 75080
{firstname.lastname}@utdallas.edu

Abstract. This paper presents ARGOS, the first system that can automatically uncover the semantics of kernel objects directly from a kernel binary. Based on the principle of data use reveals data semantics, it starts from the execution of system calls (i.e., the user level application interface) and exported kernel APIs (i.e., the kernel module development interface), and automatically tracks how an instruction accesses the kernel object and assigns a bit-vector for each observed kernel object. This bit-vector encodes which system call accesses the object and how the object is accessed (e.g., read, write, create, destroy), from which we derive the meaning of the kernel object based on a set of rules developed according to the general understanding of OS kernels. The experimental results with Linux kernels show that ARGOS is able to recognize the semantics of kernel objects of our interest, and can even directly pinpoint the important kernel data structures such as the process descriptor and memory descriptor across different kernels. We have applied ARGOS to recognize internal kernel functions by using the kernel objects we inferred, and we demonstrate that with ARGOS we can build a more precise kernel event tracking system by hooking these internal functions.

1 Introduction

Uncovering the semantics (i.e., the meanings) of kernel objects is important to a wide range of security applications, such as virtual machine introspection (VMI) [11], memory forensics (e.g., [24,13]), and kernel internal function inference. For example, knowing the meaning of the `task_struct` kernel object in the Linux kernel can allow VMI tools to detect hidden processes by tracking the creation and deletion of this data structure. In addition, knowing the semantics of `task_struct` enables security analysts to understand the set of functions (e.g., the functions that are responsible for the creation, deletion, and traversal of `task_struct`) that operate on this particular data structure.

However, uncovering the semantics of kernel objects is challenging for a couple of reasons. First, an OS kernel tends to have a large number of objects (up to tens of thousands of dynamically created ones with hundreds of different semantic types). It is difficult to associate the meanings to each kernel object when given such a large number. Second, semantics are often related to meaning, which is very vague even to human beings. It is consequently difficult to precisely define semantics that can be reasoned by a machine. In light of these challenges, current practice is to merely rely on human beings to *manually* inspect kernel source code, kernel symbols, or kernel APIs to derive and annotate the semantics of the kernel objects.

To advance the state-of-the-art, this paper presents ARGOS, the first system for *A*utomatic *R*everse en*G*ineering of kernel *O*bject *S*emantics. To have a wider applicability and practicality, ARGOS works directly on the kernel binary code without looking at any kernel source code or debugging symbols. Similar to many other data structure (or network protocol) reverse engineering systems (e.g., [5,7,17,4,18,23]), it is based on the principle of *data uses tell data types*. Specifically, it uses a dynamic binary code analysis approach with the kernel binary code and the test suites as input, and outputs the semantics for each observed kernel object based on how the object is used.

There are two key insights behind ARGOS. One is that different kernel objects are usually accessed in different kernel execution contexts (otherwise, they should be classified into the same type of object). Consequently, we can use different kernel execution contexts to classify each object. The other is that we can further derive the semantics by using well-accepted public knowledge, such as the user level system call (syscall for brevity henceforth) interface, which is used when developing user level applications; the kernel level exported API interface, which is used when developing kernel modules; and the different memory operations such as memory read and write, which we can use to track and associate the execution context with each kernel object.

To address the challenge of precisely defining the kernel object semantics, we introduce a bit vector to each kernel object. This bit vector encodes which syscalls accessed the object (one syscall per bit), and using what kind of access (e.g., create, read, write, and destroy). In total, given an N number of syscalls for a given OS kernel, our bit vector has $4N$ bits in length for each distinctive kernel object. This $4N$ bit vector captures all the involved syscalls during the lifetime of a particular kernel object, which can be understood as a piece of information contributed by the accessing syscalls. Consequently, the meaning for each object is represented by the syscalls that accessed it and the different ways that it was accessed. Such information can uniquely represent each kernel object and its meaning.

Since syscalls are usually compatible across different kernels for the same OS family, this would allow ARGOS to directly reason about the kernel objects for a large set of OSes. More importantly, it would also allow ARGOS to interpret the meaning of kernel objects in a unified way. For instance, there could be different names for certain kernel objects even though they have the same semantic type. By using the same encoding across different OSes, we can uniformly identify the common important data structures such as process descriptor, memory descriptor, file descriptor, etc., regardless of their symbol names.

There will be many valuable applications enabled by ARGOS. One use case, as we will demonstrate in this paper, is that we can use the uncovered object semantics to infer the internal kernel functions. The knowledge of the internal kernel functions is extremely useful for kernel malware defense. Other applications include kernel data structure reverse engineering, virtual machine introspection, and memory forensics. In particular, ARGOS will complement the existing data structure reverse engineering work that previously only focused on recovering the syntactic information (i.e., the layout and shape) of data structures by adding the semantic information.

In summary, we make the following contributions in this paper:

- We present ARGOS, the first system that is able to automatically uncover the semantics of kernel objects from kernel binary code.
- We introduce a bit-vector representation to encode the kernel object semantics. Such representation separates the semantic encoding and semantic presentation, and makes ARGOS work for a variety of syscall compatible OSes.
- We have built a proof-of-concept prototype of ARGOS. Our evaluation results show that our system can directly recognize a number of important kernel data structures with correct semantics, even across different kernels.
- We show a new application by applying ARGOS to discover the internal kernel functions, and demonstrate that a better kernel event tracking system can be built by hooking these internal kernel functions.

2 System Overview

While the principle of "*data uses tell data types*" is simple, we still face several challenges when applying it for the reverse engineering of kernel object semantics. In this section, we walk through these challenges and give an overview of our system.

Challenges. Since ARGOS aims to uncover kernel object semantics, we have to first define what the semantics are and how a machine can represent and reason about them. Unfortunately, this is challenging not just because the semantics themselves are vague but also because it is hard to encode.

Second, where should we start from? Given kernel binary code, we can inspect all of its instructions (using static analysis) or the execution traces (using dynamic analysis). While static analysis is complete, it is often an over approximation and may lead to imprecise results. Dynamic analysis is the opposite. Therefore, we have to make a balance and select an appropriate approach.

Third, there are various ways and heuristics to perform the reverse engineering, e.g., blackbox approaches by feeding different inputs and observing the output differences, or whitebox approaches by comparing the instruction level differences (e.g., [14]). It is unclear which approach we should use, and we have to select an appropriate one for our problem.

Insights. To address these challenges, we propose the following key ideas to reverse engineer the kernel object semantics:

- **Starting from well-known public knowledge**. Similar to many other reverse engineering systems, ARGOS must start from a well-known knowledge base to infer unknown knowledge. For a given OS kernel, there are two pieces of well-known public knowledge: (1) the syscall specification that is used by application programmers when developing user level programs, and (2) the public exported kernel API specification that is used by kernel module programmers when developing kernel drivers. Therefore, in addition to the kernel test cases, ARGOS will take syscall and kernel API specifications as input to infer the kernel object semantics.

- **Using execution context differencing**. In general, different kernel objects are usually accessed in different execution contexts (otherwise, they should be classified as the same object). Consequently, we can use different execution contexts to classify each object, and we call this approach execution context differencing.
- **Encoding the semantics with a bit-vector**. To keep a record of the different accesses by different syscalls, we use a bit-vector associated with each distinctive object. This bit-vector captures which syscall, under what kind of context, accessed the object. Through this approach, we can separate the semantic encoding and presentation.

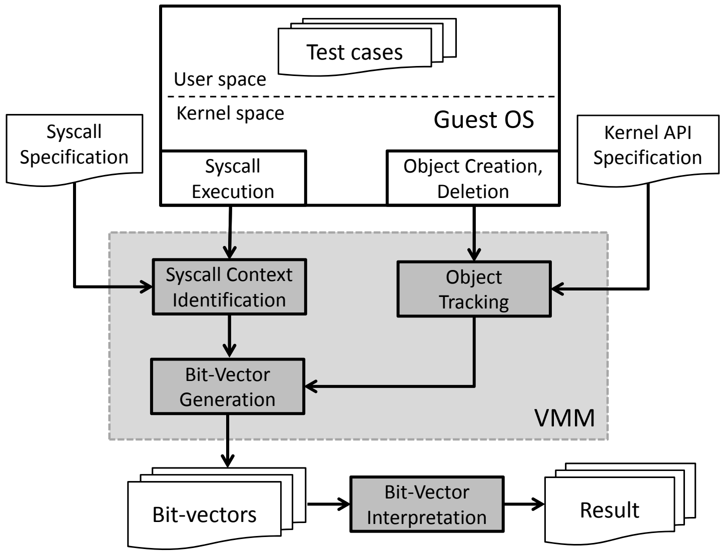

Fig. 1. An Overview of ARGOS.

Overview. To make ARGOS work with a variety of OS kernels, we design it atop a virtual machine monitor (VMM), through which we observe and trace each kernel instruction execution. As shown in Fig. 1, there are four key components inside ARGOS: *object tracking*, *syscall context identification*, *bit-vector generation*, and *bit-vector interpretation*. They work as follows: starting from kernel object creation, *object tracking* tracks the dynamically created kernel objects and indexes them based on the calling context during object creation; whenever there is an access, which is defined as a 4-tuple (create, read, write, destroy), *syscall context identification* resolves the current context and tracks which syscall is accessing the object and under what kind of context. This information will be recorded by *bit-vector generation* during the lifetime for each observed object. Finally, we use *bit-vector interpretation* to interpret the final semantics based on the encoded bit vector.

Scope and Assumptions. We focus on the reverse engineering of the object semantics of the OS kernels that are executed atop the 32-bit x86 architecture. To validate our experimental results with the ground truth, we use open source Linux kernels as the testing software. Note that even though the source code of Linux kernel is open, it is actually non-trivial to retrieve the semantic information for each kernel object. Currently, we use a manual approach to reconstruct the semantic knowledge based on our best understanding of Linux kernels, and compare with the results generated by ARGOS.

While we can integrate other techniques (e.g., REWARDS [18] and Howard [23]) to recover the kernel object syntax (i.e., the fields and layout information), we treat each kernel object as a whole in this paper and focus on the uncovering of the kernel object semantics, an important step to enable many other applications.

In addition, we assume the users of our tool will provide a syscall specification that includes each syscall number and syscall name, as well as an exported kernel API specification that includes the instruction addresses of kernel object allocation functions (e.g., `kmalloc`) such that ARGOS can hook and track kernel object creation and deletion. Meanwhile, since ARGOS needs to watch each instruction execution, we build our tool atop PEMU [26], which is a dynamic binary code instrumentation framework based on QEMU [2]. Also, we do not attempt to uncover the semantics for all kernel data, but rather focus on dynamically accessed kernel objects.

3 Design and Implementation

In this section, we present the detailed design and implementation of each component of ARGOS. Based on the flow of how ARGOS works, we first describe how we track each kernel object in §3.1; then describe how we resolve the corresponding syscall execution context when an object is accessed in §3.2; next, we present the *bit-vector generation* component in §3.3, followed by the *bit-vector interpretation* component in §3.4.

3.1 Object Tracking

Since the key idea of ARGOS is based on the object use to infer the object semantics and kernel objects are usually dynamically allocated, we have to (1) track object allocation/deallocation, (2) track the size of each object, and (3) index each object such that when given a dynamic access of the kernel object, we are able to know to which object the address belongs. In the following, we describe how we achieve these goals.

1). Tracking the Object Allocation and Deallocation. A widely used approach to track a kernel object is to hook its creation and deletion APIs. These APIs are usually publicly accessible for kernel developers (even in closed source OSes such as Microsoft Windows). In our implementation, we just hook the kernel object allocation and deallocation functions such as `kmem_cache_alloc`/`kmem_cache_free`, `kmalloc`/`kfree`, `vmalloc`/`vfree` at the VMM layer for the Linux kernel. To support efficient look up, we use a red-black (RB) tree indexed by their starting address and size to track the allocated object.

2). Tracking the Object Size. Unlike at user level, we can intercept the argument to `malloc`-family functions to identify the object size (while this is still true for `kmalloc`), but there is no size argument to many other kernel object allocation functions (e.g., `kmem_cache_alloc`). The reason is that the kernel memory allocator (e.g., the slab or slub allocator) usually caches similar size type objects and organizes them into clusters. For example, when allocating a kernel object (e.g., `task_struct`), kernel developers will just pass a flag argument and a pointer argument that points to `kmem_cache` structure, which is the descriptor of the cluster that contains the objects with similar size. This descriptor is created by the kernel API `kmem_cache_create` and the size of the object is passed to this descriptor's creation function. Then one may wonder why the size argument passed to `kmem_cache_create` cannot be used as the object size. This is because this size is actually an over approximation and the size of the real kernel object can be smaller than the one specified in the descriptor. Meanwhile, the pointer argument of `kmem_cache_alloc` can point to the `kmem_cache` that has entirely different types of objects. For instance, our trace with the slab allocator in Linux 2.6.32 shows that the kernel objects of the `file` and `vfs_mount` data structures are stored in the same `kmem_cache` even though they have different types and different sizes.

Therefore, we have to look for new techniques to recognize the kernel object size. Since we use dynamic analysis, we can in fact track the allocated object size at run time based on the object use. While this is still an approximate approach, it is at least sound and we will not make any mistakes when determining to which object a given virtual address belongs. Specifically, to access any dynamically created object, there must be a base address. Right after executing a kernel object allocation function, a base address is returned, which we shall refer to as v. Any further access to the field of the object must start from v, or the propagation of v. As such, we can infer the object size by monitoring the instruction execution and checking whether there is any memory address that is derived from the virtual address v as well as its propagation.

Without loss of generality, as shown in Fig. 2 (a), when an object O_i is created, we will have its starting (i.e., base) address v (suppose it is stored in `eax`). To access the fields of O_i, there must be a data arithmetic operation of the base pointer (or its derivations), and we can therefore infer the size based on the offset of the access. For instance, as shown in Fig. 2 (a), assume the kernel uses `eax`$+m$ to access a field p of O_i, then we can get the size of O_i as $m+4$ from this particular operation. Then, assume the kernel inserts O_i to some other data structures (e.g., a linked list); it must compute a deref-

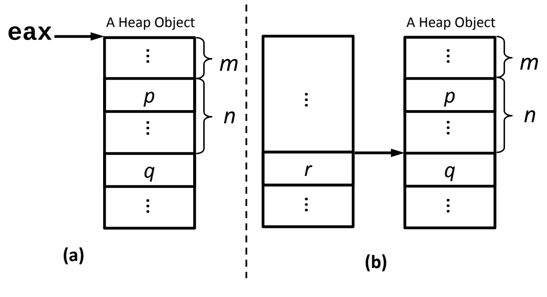

Fig. 2. An Example Illustrating How to Track the Object Size. Note that Taint (eax) = Taint (p) = Taint (q) = Taint (*r) = T_i, and Taint (r) = T_j.

erenced address of O_i such that traversing other objects can reach O_i. Assume this address is q, which is computed from $p + n$, then we can infer the $Size(O_i)$ as $(m + n + 4)$ according to the execution of these accesses. Next, assume we assign the address of q to r (Fig. 2 (b)). Then all future dereferences will use $*r$ as a base address to access O_i (instead of v, the starting address), and we can similarly derive the size based on the pointer arithmetic. Note that when dereferencing a kernel object, the kernel can start from its middle instead of the starting address, which is very common in both the Windows and Linux kernels.

Therefore, in order to resolve the size, we have to know that `eax`, p, q, and $*r$ actually all reference the same allocated object (i.e., they belong to the same closure). If we assign a unique taint tag for each O_i using T_i, namely Taint(`eax`)=Taint(v)=T_i, then we can propagate T_i to $p, q, *r$ based on the pointer data movement and arithmetic operations. Thus, this eventually leads to a dynamic taint analysis [19] approach to decide whether `eax`, p, q, and $*r$ belong to the same T_i. Since taint analysis is a well established approach, we omit its details for brevity in this paper. Basically, in our taint analysis, we capture how a memory access address is computed from the base address v and its propagations (e.g., `eax`), from which we resolve the object size. This size is the one being observed at run time.

Meanwhile, kernel objects usually point to each other. Looking at the point-to graph can facilitate the inference of the important kernel data structures based on their relations. Since we have assigned a unique taint tag T_i for each kernel object, we can now track the dependence between kernel objects by looking at their taint tags during memory write operations. Specifically, whenever there is a memory write, we will check the taint tags of both its source and destination operand. If they belong to our tracked objects, we will connect these two objects using their point-to relation and store this information in their static object types. The particular offset for the two objects of the point-to relation will also be resolved. This information, namely object O_i at offset k points to O_j at offset l, will be recorded.

3). Indexing the Dynamic Kernel Objects with Static Representation. Since kernel data structure semantics are static attributes, they should be applied to all of the same type of a kernel object. However, when we use dynamic analysis, what we observe is instances of kernel objects. Therefore, we need to translate these dynamic instances into static representations such that our bit-vector can just associate with the static representation instead of the dynamic object instances.

In general, there are two basic approaches when converting dynamic object instances into static forms: (1) using the concatenation of all the call-site addresses from the top callers to the callee, or (2) using the program counter of the instruction that calls a kernel object allocation function. The first approach can capture all the distinctive object allocations, but it may over classify the object types since the same type can be allocated in different program calling contexts. While the second approach mitigates this problem, it cannot handle the case where an allocation function is wrapped. Therefore, the solution is always domain-specific and somewhere in-between of these two approaches.

In our design, we adopt the second approach because we observe that a single kernel object can often be allocated in different calling contexts (e.g., we observe that the task_struct in Linux can be allocated in syscalls such as vfork, clone, etc). If we assign the call-site chain as the static type, we could over classify the kernel object (having an N-to-one mapping). Also, our analysis with a ground-truth labeled Linux kernel 2.6.32 shows that when we use the call site PC of the allocation function (denoted as $PC_{kmalloc}$) to assign the static type, 80.3% of the kernel objects have a one-to-one mapping. In contrast, when we tried the call-site chain approach, we found 97.5% of the objects had N-to-one mapping. Therefore, eventually, in our current design, we decided to take the second approach.

Summary. In short, our *object tracking* component will track the lifetime of the dynamically allocated object using an RB-tree, which we call an RB$_{type}$ tree. It is used to store <v, s, T_i, $PC_{kmalloc}$>, which is indexed by v, where v is the starting address, s is the current resolved size (subject to be updated during run-time), T_i is the taint tag for O_i, and $PC_{kmalloc}$ is the static type of the allocated object. The reason to use the RB$_{type}$-tree is to speed up locating the static type (encoded by $PC_{kmalloc}$) when given a virtual address, and we maintain an RB$_{type}$-tree to track these dynamically allocated objects.The basic algorithm is to check whether a given virtual address α falls into $[v, v + s]$ of our RB-tree node; if so, we return its $PC_{kmalloc}$ as the type. Also, we maintain a hash table (HT) that uses $PC_{kmalloc}$ as the index key. This HT will be used to store the bit-vectors of the kernel objects based on their assigned static types as well as the point-to relations between objects.

3.2 Syscall Context Identification

To associate the execution context with each dynamically accessed kernel object, we must resolve the execution context when an instruction is accessing our monitored object. The execution context in this paper is defined as *the information that captures how and when a piece of data gets accessed*. More specifically, as our starting point of the known knowledge is the syscall, we need to first resolve which syscall is currently accessing a given piece of data. Also, since we need to capture the different data accesses in order to identify the internal functions (e.g., the internal function that creates the process descriptor structure), we have to further classify the data access into different categories such as whether it is a read access or a write access.

Precisely Identifying the Syscall Execution Context. When a given kernel object is accessed, we need to first determine which syscall is accessing it. Since an execution context must involve a stack (to track the return addresses for instance), we can use each kernel stack to differentiate each execution context. Whenever there is a kernel stack change, there must be an execution context change.

Then how many kernel stacks are inside an OS kernel at any given moment? This depends on how many user level processes and kernel level threads are running. In particular, each user level process (including user level threads) will have a corresponding kernel stack. This kernel stack is used to store the local variables and return addresses when a particular syscall is executed inside the kernel. Besides the kernel stack for user level processes to execute syscalls, there are also kernel threads that are responsible for handling background tasks such as asynchronous event daemons (e.g., ksoftirqd) or worker threads (e.g., pdflush, which flushes dirty pages from the page cache). The difference between kernel threads and user level processes is that kernel threads do not have any user level process context and will not use the syscall interface to request kernel services (instead they can directly access all kernel functions and data structures).

Therefore, by tracking each syscall entry and exit (e.g., sysenter/sysexit, int 0x80/iret) and stack change (e.g., mov stack_pointer, %esp), we can identify the syscall execution context, as demonstrated in our earlier work VMST [10]. Note that the execution of the top half of an interrupt handler may use the current process' or kernel thread's kernel stack, and we have to exclude this interrupt handler's execution context. Fortunately, the starting of the interrupt handler's execution can be observed by our VMM, and these handlers always exit via iret. As such, we can precisely identify the interrupt execution contexts and exclude them from the syscall context.

To resolve to which syscall the current execution context belongs, we will track the syscall number based on the eax value when the syscall traps to the kernel for this particular process. The corresponding process is indexed by the base address of each kernel stack (not the CR3 approach as suggested by Antfarm [15] because threads can share the same CR3). We use the 19 most significant bits (MSB) of the kernel esp, i.e., the base address of the stack pointer (note that the size of Linux kernel stack is 8192=2^{13} bytes), to uniquely identify a process. The base address of the stack pointer is computed by monitoring the memory write to the kernel esp. We also use an RB-tree, which we call RB_{sys} tree, to dynamically keep the MSB19(esp) and the syscall number from eax for this process such that we can quickly return the syscall number when given a kernel esp.

Tracking Syscall Arguments of Interest. The majority of syscalls are designed for a single semantic goal such as to return a pid (getpid) or to close a file descriptor. However, there are syscalls that have rich semantics—namely having different behaviors according to their arguments. One such a syscall is sys_socketcall, which is a common kernel entry point for the socket syscall. Its detailed argument decides which particular socket function to be executed (e.g., socket, bind, listen, setsockopt, etc.). Therefore, we have to parse its arguments and associate the arguments to the syscall context such that we can infer the exercised kernel object semantics under this syscall.

Besides sys_socketcall, in which we have to track its arguments, we find two other syscalls (sys_clone and sys_ unshare) that also have strong argument controlled behavior. In particular, sys_clone can associate certain important kernel objects with the new child process when certain flags are set (e.g., CLONE_FS flag will make the caller and the child process share the same file system information), and sys_unshare can reverse the effect of sys_clone by disassociating parts of the process execution context (e.g., when CLONE_FS is set, it will unshare file system attributes such that the calling process no longer shares its root directory, current directory, or umask attributes with any other processes). Therefore, we will track these three syscalls, and associate their arguments with the exercised kernel objects, because these arguments specify the distinctive kernel behavior of the corresponding syscall.

3.3 Bit-Vector Generation

Having tracked all dynamically allocated objects that are executed under each specific syscall execution context, we will then attach this context using a bit-vector to the object type we resolved in *object tracking*. The length of our bit-vector is 4*N bits, where N is the number of syscalls provided by the guest OS kernel. Meanwhile, for each syscall, we will track and assign the following bits in the bit-vector to 1 or 0 based on:

– C-**bit**: whether this syscall created the object;
– R-**bit**: whether this syscall read the object;
– W-**bit**: whether this syscall wrote the object ;
– D-**bit**: whether this syscall destroyed the object.

These bits together form an entropy of how a syscall uses the object, from which we can derive the meanings.

Since our *bit-vector generation* is the core component in our system and it connects the *object tracking* and *syscall context identification* components, in the following we present a detailed algorithm to illustrate how it exactly works. At a high level, we use an online algorithm to resolve the object's static type, syscall context, and different ways of access, and generate the bit vector, which is stored in a hash table indexed by the object's static type (i.e., $PC_{kmalloc}$). As presented in Algorithm 1, each kernel instruction execution is monitored in order to resolve and generate our bit vector.

In particular, before beginning our analysis, we will first create a hash table (HT) at line 2 that stores the bit vector of the accessed kernel object. Then we iterate through each kernel instruction (line 3-43). We first check whether the current instruction involves pointer data arithmetic (line 5-6), if so, we will track the dependences of the involved pointers and infer their sizes. Next, we check if the instruction is `sysenter/int0x80` (line 8-10). If so, we update the syscall context tracking data structure RB$_{sys}$-tree that stores the syscall number for the current process, which is determined by variable Ex. This Ex is a global variable, which keeps the MSB19(`esp`) and gets updated when kernel

Algorithm 1: Bit-vector Generation

```
1  Procedure BvG () :
   Output: Hash Table $HT$ that contains bit vector of the
   observed kernel object type and their point-to relations.
2  $HT \leftarrow$ CreatSemanticTypeBitVectorHashTable() ;
3  for each executed instruction $I$ do
4      $op \leftarrow Operand(I)$;
5      if PointerArithmeticOrPropagation(op) then
6          TaintOPAndUpdateTrackedSize(op);
7      switch $I$ do
8      case sysenter/int0x80
9          UpdateRBsysNode($Ex$, eax)    ;
10         // $Ex$ represents the process context
11     case syscall(exit_group) SUCCESS
12         RemoveRBsysNode($Ex$)
13     case mov op, esp
14         $Ex \leftarrow$ MSB19(esp);
15         InsertRBsysNodeIfNotExist($Ex$)
16     case $PC_{kmalloc}$: eax ← call {kmalloc}
17         $t \leftarrow PC_{kmalloc}$;
18         $T_i \leftarrow$ GetUniqueTaintTag();
19         InsertRBtypeNode(eax, 4, $T_i$, $t$);
20         InsertHTifNotExist($t$);
21         if $I \in SyscallContext$ then
22             sysnum $\leftarrow$ QueryRBsysNum($Ex$);
23             $HT$[t][sysnum][C-bit] $\leftarrow 1$;
24     case $PC_{kfree}$: call {kfree}(v)
25         $t \leftarrow$ QueryRBtype(v);
26         RemoveRBtypeNode($v$);
27         if $I \in SyscallContext$ then
28             sysnum $\leftarrow$ QueryRBsysNum($Ex$);
29             $HT$[t][sysnum][D-bit] $\leftarrow 1$;
30     case MemoryAccess(op)
31         if $I \notin SyscallContext$ then
32             continue;
33         switch access do
34             $src \leftarrow Source(op)$;
35             $t \leftarrow$ QueryRBtype(src);
36             sysnum $\leftarrow$ QueryRBsysNum($Ex$);
37         case READ
38             $HT$[t][sysnum][R-bit] $\leftarrow 1$;
39         case WRITE
40             $HT$[t][sysnum][W-bit] $\leftarrow 1$;
41             $dst \leftarrow Destination(op)$;
42             TrackingObjectPointToRelation(HT,
43                                   $src$, $dst$);
44 return $HT$;
```

stack switches (line 13-15). The node of the RB$_{sys}$-tree will be deleted when the process exits (line 11-12).

Next, when the kernel execution is to create an object (line 16-23), we then insert the created instance into the RB$_{type}$-tree that keeps the type and size information about the object (line 19). We also insert the static type assigned for this object (namely $PC_{kmalloc}$) into the HT if this type has not been inserted before (line 20). In addition, we update the bit vector with a C-bit for this particular object if the object is created under the syscall execution context (line 21-23), neither in top-half nor bottom-half. Similarly, we remove the dynamic instance from the RB$_{type}$-tree, and update the D-bit in the corresponding HT entry if the necessary, when the object is deallocated (line 24-29). Then, if the instruction is accessing the memory address that belongs to our tracked kernel object (line 30) and is under a syscall execution context (line 31-42), we update the corresponding R-bit and W-bit based on the access (line 38, 40). We also track the object point-to relation if there is a memory write that involves two monitored kernel objects (line 42). All the involved data structures are presented in Fig. 3.

3.4 Bit-Vector Interpretation

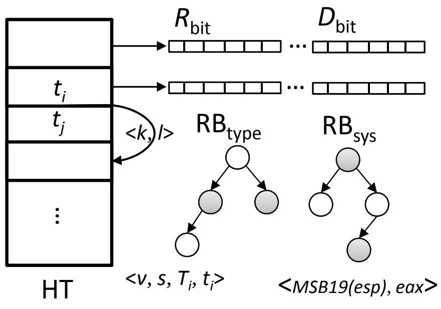

Fig. 3. The Data Structures Used in ARGOS.

Having generated the bit-vector for each observed object type, ARGOS is then ready to finally output the meanings (i.e., semantics) of the observed objects. Since our bit-vector has 4*N bits in length, it contains a very large amount of information, sufficiently distinguishing each different semantic type. In particular, our bit-vector captures how a syscall accessed the object during the life time of the object. Such an access denotes the connection between the object and the syscall. At a high level, we can view the bit-vector as representing (1) which of the syscalls have contributed to the meaning of the object, (2) how these syscalls contributed (recorded in our R,W,C,D-bits). Given such rich information, there could be many different approaches to derive the semantics and interpret the meanings.

One possible approach is to simply transform the bit-vector to a large integer value (using a deterministic algorithm), and map the integer value to a kernel object acquired from the ground truth. If there is always a one-to-one mapping, then this approach would work. For instance, from the general OS kernel knowledge, we know that a process descriptor (i.e., `task_struct` in Linux), is usually the root of the kernel data structure when accessing all other objects inside OS kernel for a particular process. Many of the syscalls would have accessed this object. Therefore, a process descriptor would have a larger value than many other data structures when translating these bits into integers. Based on such values, we could possibly determine the semantic types.

In our current ARGOS design, we present another simple approach: instead of checking all bit-vectors (normalizing them to an integer value), we check certain syscalls for the object of our interest from the bit-vector, by using the manually derived rules based on general syscall and kernel knowledge. Again, taking `task_struct` as an example, we know that this data structure must be created by a `fork`-family syscall, and destroyed by a `exit_group` syscall. When there is a `getpid` syscall executed, it must first fetch this data structure, from which to traverse other data structures to reach the `pid` field. Therefore, we can develop data structure specific rules to derive the semantics by checking the bit-vectors. We have developed a number of such rules to recognize the important kernel data structures as presented in §4.

4 Evaluation

In this section, we present how we evaluate ARGOS to uncover the object semantics. We first describe how we set up the experiment in §4.1, and then present our detailed results in §4.2.

4.1 Experiment Setup

Since we focus on the reverse engineering of the kernel object semantics, we have to compare our result with the ground truth. To this end, we took two recently released Linux kernels: `Linux-2.6.32` and `Linux-3.2.58`, running in `debian-6.0` and `debian-7`, respectively, as the guest OS for ARGOS to test. Each guest OS is configured with 2G physical memory. The main reason to use the open source Linux kernel is because we can have the ground truth. Therefore, in our object tracking, we also keep the truth type when the object is created in our object tracking. The truth type is acquired through a manual analysis of the corresponding kernel source code. The host OS is `ubuntu-12.04` with kernel `3.5.0-51-generic`. The evaluation was performed on a machine with an Intel Core i-7 CPU and 8GB physical memory.

An end user needs to provide three pieces of information to ARGOS as input: a syscall specification, a kernel API specification, and the test cases.

- **Syscall Specification**. Basically, it just needs the syscall number and the corresponding syscall name. In addition, it also requires an understanding of the arguments and corresponding semantic behavior of three syscalls (`sys_socketcall`, `sys_clone` and `sys_unshare`), which are used to derive the semantics of the objects accessed in these syscalls.
- **Kernel API Specification**. To track the dynamic object creation and deletion, we need the Kernel API specification of the `kmalloc` family of functions. Similar to the syscall specification, we just need the name of each function, its starting virtual address, and its arguments such that we can intercept these function executions.
- **Test Cases**. ARGOS is a dynamic analysis based system. We need to drive the kernel execution through running the test cases. Ideally, we would like to use existing test cases. To this end, we collected several user level benchmarks including `ltp-20140115` and `lmbench-2alpha8`. We also used all the test cases from the Linux-test-project [1].

4.2 Detailed Result

Rule Num	Detailed Rules	Data Structure
I	`sys_clone`[C] ∩ `sys_getpid`[R]	`task_struct,pid`
II	((`sys_clone`[C] - `sys_vfork`[C]) ∩ `sys_brk`[RW]) ∩ `sys_munmap`[D]	`vm_area_struct`
III	((`sys_clone`[C] - `sys_vfork`[C]) ∩ `sys_brk`[RW]) - `sys_munmap`[D]	`mm_struct`
IV	`sys_open`[C] ∩ `sys_lseek`[W] ∩ `sys_dup`[R]	`file`
V	`sys_clone`[C] - `sys_clone`[C](CLONE_FS)	`fs_struct`
VI	`sys_clone`[C] - `sys_clone`[C](CLONE_FILES)	`files_struct`
VII	`sys_mount`[C] ∩ `sys_umount`[D]	`vfs_mount`
VIII	`sys_socketcall`[C](SYS_SOCKET) ∩ `sys_socketcall`[W] (SYS_SETSOCKOPT)	`sock`
IX	`sys_clone`[C] - `sys_clone`[C](CLONE_SIGHAND)	`sighand_struct`
X	`sys_capget`[R] ∩ `sys_capset`[W]	`credential`

Table 2. The Inference Rules We Developed to Recognize The Semantics of Important Kernel Data Structures.

In total, it took ARGOS 14 hours[1] each to run all the test programs (the most time consuming part is the LTP test cases) for the testing guest OS, with a peak memory overhead of 4.5G at the host level for the 2G guest OS. Specifically, we observed 105 static types for `Linux-2.6.32`, and 125 for `Linux-3.2.58`. Due to space limitations, we cannot present the detailed representation of the bit-vectors for all these objects, and instead we just present the statistics of their bit vectors.

Syscall Type	Short Name	#Syscalls	
		Linux-2.6.32	Linux-3.2.58
Process	P	90	92
File	F	152	156
Memory	M	19	21
Time	T	13	13
Signal	G	25	25
Security	S	3	3
Network	N	2	4
IPC	I	7	7
Module	D	4	4
Other	O	3	3
Total	-	317	328

Table 1. Syscall Classification

We first categorized the syscalls into groups based on the different type of resources (e.g., processes, files, memory, etc.) that the syscalls aim to manage. The classification result is presented in Table 1. We can notice that these two kernels do not have the exact same number of syscalls, and `Linux-3.2.58` introduces 11 additional syscalls to `Linux-2.6.32`. Consequently, the length of their bit-vectors are different. We present a number of bit-vector statistics in the last 10 columns of Table 3. The statistics of these bit vectors show the distributions of the sycalls that have read and write access of each corresponding object. For instance, for the `pid` data structure presented in the first row, its P=25 means there are 25 process related syscalls that have accessed this object.

Next, we present how we would discover the semantics of each kernel object. As discussed in §3.4, there could be several different ways of identifying the kernel objects and their semantics. In the following, we demonstrate a general way of identifying the kernel objects that are of security interest (such as process descriptor, memory descriptor, etc.) by manually developing rules based on the semantics of the syscalls (which is generally known to the public) and also using execution context differencing. In total we developed 10 rules, which are presented in Table 2.

[1] Note that ARGOS is an *automated* offline system. Performance is not a big issue as long as we produce the result in a reasonable amount of time.

Rule Num	Kernel Version	Static Type	Symbol Name	Traced Size	Statistics of the R/W Bit Vector									
					P	F	M	T	G	S	N	I	D	O
I	2.6.32	`c10414e8`	`pid`	44	25	16	4	0	3	0	1	3	1	0
		`c102db48`	`task_struct`	1072	47	48	5	0	12	0	1	1	2	0
	3.2.58	`c104bb18`	`pid`	64	28	24	3	0	3	0	1	3	1	0
		`c10371e3`	`task_struct`	1072	73	109	13	6	19	1	2	7	2	0
II	2.6.32	`c102d8af`	`vm_area_struct`	88	4	17	12	0	3	0	0	1	1	0
	3.2.58	`c1036f6a`	`vm_area_struct`	88	3	5	12	0	0	0	1	1	1	0
III	2.6.32	`c102d762`	`mm_struct`	420	15	6	5	0	0	0	0	1	1	0
	3.2.58	`c1036dc8`	`mm_struct`	448	15	9	6	0	0	0	1	1	1	0
IV	2.6.32	`c10b23ae`	`file`	128	41	93	12	0	10	0	1	7	2	0
	3.2.58	`c10ceea4`	`file`	160	35	97	12	0	11	0	1	7	2	0
V	2.6.32	`c10cac66`	`fs_struct`	32	4	50	0	0	0	0	1	1	1	0
	3.2.58	`c10eaad7`	`fs_struct`	64	4	51	0	0	0	0	1	1	1	0
VI	2.6.32	`c10c185c`	`files_struct`	224	11	73	3	0	4	0	1	6	1	0
	3.2.58	`c10df2cd`	`files_struct`	256	39	84	5	0	6	0	1	6	1	0
VII	2.6.32	`c10c3a4c`	`vfs_mount`	128	1	17	0	0	0	0	0	0	1	0
	3.2.58	`c10dfd37`	`vfs_mount`	160	3	4	0	0	0	0	0	0	1	0
VIII	2.6.32	`c11cd7c8`	`sock`	1216	19	55	8	0	9	1	6	6	2	0
	3.2.58	`c11cd7c8`	`sock`	1248	28	74	7	0	9	1	1	6	2	0
IX	2.6.32	`c102dfd8`	`sighand_struct`	1288	15	5	0	0	12	0	1	1	1	0
	3.2.58	`c10376a7`	`sighand_struct`	1312	15	7	0	0	12	0	1	1	1	0
X	2.6.32	`c1047938`	`cred`	128	51	72	8	3	3	1	2	4	2	0
	3.2.58	`c1052611`	`cred`	128	53	75	7	3	2	1	2	4	2	0

Table 3. The Inference of the Selected Kernel Data Structures and The Statistics of Their Bit-Vector.

We tested our rules against both `Linux-2.6.32` and `Linux-3.2.58`, for which we have the manually obtained ground truth. We show that we can successfully pinpoint 11 kernel objects (presented in the 3^{rd}-column with the ground truth shown in the 4^{th}-column in Table 3) and their meanings. By using the rules we derived, there is not even a need to train for each kernel and we just use them to scan the bit-vector. In the following, we describe how we derived these rules, and how we applied them in finding the semantics of kernel objects of our interest.

Process Related. The most important process related data structure is the *process descriptor* (i.e., `task_struct` in Linux), which keeps a lot of information regarding the resources a process is using, and how to reach these resources. Surprisingly, by looking at the bit-vectors of all the kernel objects, it is actually quite simple to identify the process descriptor.

Specifically, since `task_struct` must be created by process creation related syscalls (e.g., `sys_clone`, `sys_vfork`), we can in fact scan the C-bit of the objects and check the ones that are created under these syscalls (e.g.,`sys_clone`[C]); `task_struct` must exist in this set. However, during the process creation, it will also create many other data structures such as the *memory descriptor* for this process. Consequently, we have to exclude these data structures. Our insight is that we can perform a set intersection (basically execution context differencing) to identify the desired object. Back to the process descriptor example, from a general understanding of the syscall semantics, we know that `sys_getpid` must access the `task_struct` in order to get the `pid`. As such, we can then check the `sys_getpid`[R] bit.

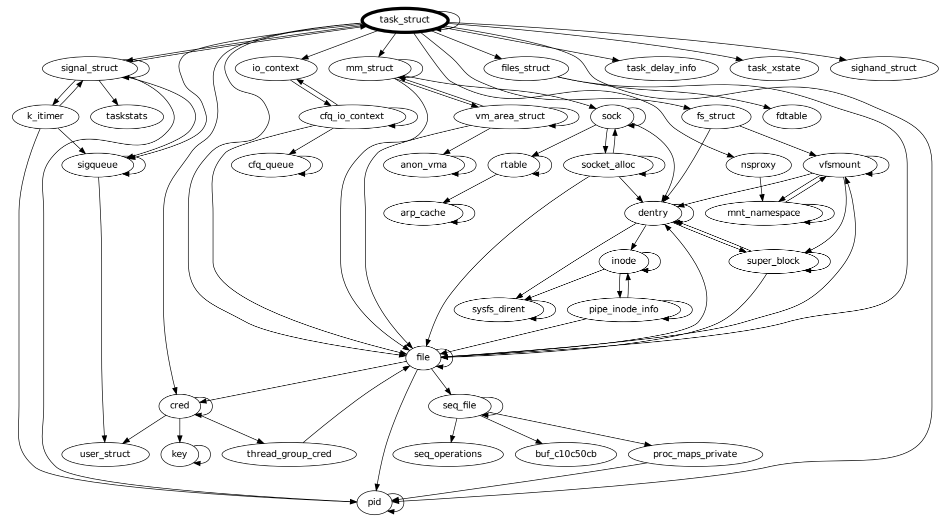

Fig. 4. The reverse engineered data structure type graph with `task_struct` as the root of Linux Kernel 2.6.32. Each node represents a reverse engineered data structure (the symbol name is just for better readability), and each edge represents the point-to relation between the data structures. There can be multiple point-to edges between two nodes at different offsets. They are merged for better readability.

Therefore, through the intersection of $\texttt{sys\_clone}[C] \cap \texttt{sys\_getpid}[R]$ (as illustrated in the first rule of Table 2), we can get two objects with static types of `c10414e8` and `c102db48`. Then the next question is how to get the `task_struct`. In fact, we can look at the relation between the data structure (i.e., the type graph we extracted). As illustrated in Fig. 4, we can see clearly that `c10414e8`[2] is reached from `c102db48`. Therefore, we can conclude that `c102db48` is the `task_struct` based on general OS kernel knowledge, and `c10414e8` is the `pid` descriptor. This rule also applies to `Linux-3.2.58`, and we can correctly recognize its `task_struct` and `pid` descriptor without any training.

Memory Related. There are two important data structures that describe the memory usage for a particular process. One is the *memory descriptor* (`mm_struct` in Linux) that contains all the information related to the process address space, such as the base address of the page table and the starting address of process code and data. Also, since memory is often divided into regions to store different types of process data (e.g., memory mapped files, stacks, heaps, etc.), the kernel uses the other important data structure called *virtual memory area descriptor* (`vm_area_struct`) to describe a single memory area over a contiguous interval in a given address space. Certainly, `vm_area_struct` can be reached from the `mm_struct`.

[2] Note that we show the symbol name instead of the address in Fig. 4 just for the readability of the type graph.

To recognize these two data structures, we again use general OS and syscall knowledge. In particular, we know that all the child threads share the same virtual space. Therefore, mm_struct and some vm_area_struct should not be created when a new thread is forked. We can then scan the C-bit of the object bit-vector of sys_clone, which is used to create the process, and sys_vfork, which is used to create threads. Then we can find a set of objects. Then, from general knowledge we know sys_brk (which changes program data segment size) will access mm_struct and some vm_area_struct. Meanwhile, sys_munmap, which aims to delete a memory region, will certainly delete vm_area_struct (then we look at its D-bit). As such, we have developed Rule-II and Rule-III presented in Table 2 to successfully identify vm_area_struct and mm_struct, respectively. Meanwhile, as illustrated in Fig. 4, we can also observe the type graph to infer the vm_area_struct because it has to be reached from mm_struct when we only look at these two data structures, which are in {(sys_clone[C] - sys_vfork[C]) ∩ sys_brk[RW]}.

File Related. There are several important file related data structures of our interest. Specifically, we are interested in the (1) *file descriptor* (file structure in Linux) that describes a file's properties, such as its opening mode, the position of the current file cursor, etc., (2) *file system descriptor* (fs_struct) that describes the file system related information including such as the root path of the file system, and the current working directory, (3) files_struct that describes the opening file table (e.g., the file descriptor array), (4) vfs_mount that describes the mounted file systems, and (5) sock structure that describes a network communication point in the OS kernel. In the following, we discuss how we recognize these data structures.

– *File Descriptor*. From general OS knowledge, we know that a file descriptor is created by the open syscall. However, open will create many other objects as well (e.g., we found 43 different types of objects by scanning sys_open[C] bits). Fortunately, we also know that lseek will definitely modify file (i.e, sys_lseek[W] will be set). Meanwhile, sys_dup will also absolutely read and write to the file structure. Therefore, we developed our Rule-IV by using sys_open[C] ∩ sys_lseek[W] ∩ sys_dup[R] to directly pinpoint the file structure.
– *File System Data Structure*. From the syscall specification, we know that when a child process is created, it will inherit many important kernel objects from its parent process. File system structure is definitely one of them. Also, sys_clone will provide flags to allow programmers to control whether to inherit or not. Recall in §3.2, we have tracked the flag of sys_clone, and we can therefore trivially identify the fs_struct. In particular, flag CLONE_FS will let the child process clone from its parent FS (it means no new fs_struct will be created). By performing sys_clone [C] - sys_clone[C](CLONE_FS), we directly pinpoint fs_struct.
– *Open File Table Structure*. Each process has its own opened file set. This is maintained by its files_struct in the Linux kernel. Similar to fs_struct identification, we check the flag CLONE_FILES of sys_clone to identify this structure, as presented in the Rule-VI in Table 2.

– *Mounting Point Descriptor*. When a file system is mounted, the OS uses a mounting point descriptor to track the mounted file system. There are two syscalls (sys_mount and sys_umount) involved in the mouting and unmouting operation. Basically, sys_mount creates a mouting structure and sys_umount removes it. To identify vfs_mount is actually quite simple, and we just perform a sys_mount $[C] \cap$ sys_umount$[D]$, which directly produces the desired data structure.
– *Socket*. By associating the argument with the syscall context for sys_socketcall, we can easily identify the socket data structure in the OS kernel. For instance, we can check the created object in sys_socketcall[C](SYS_SOCKET), and check the updated object in sys_socketcall[W][SYS_SETSOCKOPT]. An intersection of these two sets will directly identify the socket data structure.

Signal Related. Among the signal related data structures, the signal handler is of our interest since it can be subverted. To identify this data structure, it is also quite simple, especially if we check the flags of sys_clone. In particular, there is a CLONE_ SIGHAND flag, and if it is set, the calling process and the child process will share the same table of signal handlers. Thus, sys_clone$[C]$ - sys_clone$[C]$(CLONE_ SIGHAND) directly identifies the signal handler data structure. There are also other ways to identify this data structure. For instance, sys_alarm will set an alarm clock for delivery of a signal, which will modify the signal handler. By looking at sys_alarm[W] as well as the type graph, we can also easily tell the sighand_struct.

Credential Related. Each process in a modern OS has certain credentials, such as its uid, gid, euid, and capabilities used for access control. The Linux kernel uses a cred data structure to store this information. To identify this data structure, we found that sys_capget and sys_capset will set and get the cred field of a process. Both syscalls will read and write the credential objects and we can use set intersection to find this data structure (i.e., sys_capget$[R] \cap$ sys_capset$[W]$). Also, this data structure is reachable from task_struct as well.

5 Applications

Uncovering the semantics of kernel objects will be useful in many applications. For example, it allows us to understand what the created objects are in an OS kernel when performing introspection, and we can also use data structure knowledge to recognize the internal kernel functions. In the following, we demonstrate how we can use ARGOS to identify the internal kernel functions, especially the object creation and deletion functions, which is important to both kernel rootkit offense and defense.

Recently, kernel malware has been increasingly using internal functions to perform malicious actions. This is no surprise since kernel malware can call any internal kernel function because they share the same address space. For instance, prior studies have shown that instead of calling NtCreateProcess, kernel malware will directly call PspCreateProcess. Therefore, hooking these internal functions is very important to detect malware attacks. Note that PspCreateProcess is for the Windows kernel; the corresponding one in Linux is actually copy_process [9]. Then can we automatically identify these internal functions, such as copy_process?

Type	Version	Creation Function		Deletion Function	
		PC	Symbol	PC	Symbol
pid	2.6.32	c10414d0	alloc_pid	c10413de	put_pid
	3.2.58	c104bb02	alloc_pid	c104b969	put_pid
task_struct	2.6.32	c102daaf	copy_process	c102da55	free_task
	3.2.58	c103719d	copy_process	c10368a7	free_task
vm_area_struct	2.6.32	c102d730	dup_mm	c109d387	remove_vma
	3.2.58	c1036d97	dup_mm	c10b13d7	remove_vma
mm_struct	2.6.32	c102d730	dup_mm	c102d3dc	__mmdrop
	3.2.58	c1036d97	dup_mm	c1036a58	__mmdrop
file	2.6.32	c10b230d	get_empty_filp	c10b2030	file_free_rcu
	3.2.58	c10cee78	get_empty_filp	c10ceba0	file_free_rcu
fs_struct	2.6.32	c10cac50	copy_fs_struct	c10cae5b	free_fs_struct
	3.2.58	c10eaac4	copy_fs_struct	c10eaa55	free_fs_struct
files_struct	2.6.32	c10c1839	dup_fd	c1030a32	put_files_struct
	3.2.58	c10df2ab	dup_fd	c103b16d	put_files_struct
vfs_mount	2.6.32	c10c3a35	alloc_vfsmnt	c10c30ba	free_vfsmnt
	3.2.58	c10dfd23	alloc_vfsmnt	c10dfe36	free_vfsmnt
sighand_struct	2.6.32	c102daaf	copy_process	c102d148	__cleanup_sighand
	3.2.58	c103719d	copy_process	c103717b	__cleanup_sighand
sock	2.6.32	c11cd7a5	sk_prot_alloc	c11cc884	__sk_free
	3.2.58	c12146e5	sk_prot_alloc	c1214d46	__sk_free
cred	2.6.32	c1047923	prepare_creds	c1047d00	put_cred_rcu
	3.2.58	c10525fe	prepare_creds	c105239b	put_cred_rcu

Table 4. Internal Kernel Function Recognization for the Testing Linux Kernels.

Fortunately, it is quite straightforward to identify some of the internal functions given the semantics of the identified kernel data structure. Take `task_struct` as an example: once we have understood a dynamically allocated object is a `task_struct`, we can check which function calls the object allocation for `task_struct`, and the caller is usually the one that is responsible for the object creation. Interestingly, while this is a simple heuristic, we tested with the two kernels and found it works well. Therefore, for this experiment, we also instrumented the kernel execution and tracked the call-stacks such that we can identify the parent function of our interest.

Based on the above heuristic, we have applied ARGOS to recognize the creation and deletion functions for the objects we identified in Table 3. This result is presented in Table 4. Again, there is no false positive while using this very simple caller-callee heuristic, and we correctly identified these functions when compared with the ground truth result in the kernel source code. For readability, we present the corresponding symbols of these functions, in addition to the PCs that denote their starting addresses.

For proof-of-concept, we then developed a virtual machine introspection [11] tool atop QEMU to track and interpret the kernel object creation and deletion events related to the object we reverse engineered by hooking the internal kernel functions listed in Table 4. Without any surprise, our tool can successfully track the corresponding events for all process creations, including even a hidden process that is created with the internal function by a rootkit we developed.

6 Limitations and Future Work

While we have demonstrated we can infer the kernel object semantics from the object use, there are still a number of avenues to improve our techniques. In the following, we discuss each of the limitations of our system and shed light on our future work.

First and foremost, we need to develop more rules or other approaches to derive the kernel object semantics based on the bit-vectors. Currently, we just illustrated we can recognize some of the kernel data structures through syscall execution context diffing and general OS knowledge. As discussed earlier in §3.4, there could be many other alternatives, such as assigning different weights to the bit of interest and then converting the bit-vector into a numeric value, from which semantics could be mapped. Meanwhile, there might also be an interesting solution of only tracking a certain number of syscalls instead all of them. We leave the validation of these alternative approaches to one of our future works.

Second, currently ARGOS only aims to reveal the semantics of the kernel data structures; it does not make any effort to reveal the syntax (especially the layout of each data structure) or meaning for each field of each data structure. Since there are several existing efforts (e.g., [18,16,23,27]) focusing on user level data structure reverse engineering, especially on field layout and syntax, we plan to integrate these techniques into ARGOS to give it more capabilities.

Third, ARGOS will have false negatives because of the nature of dynamic analysis. In addition, it will not be able to track an object if its allocation functions are inlined since it uses the dynamic hooking mechanism to intercept `kmalloc` family functions. To identify these inlined kernel object allocation and deallocation functions would require a static analysis of the kernel binary code, and we leave it to another of our future works. Also, our current design uses $PC_{kmalloc}$ to type the kernel object, but there are still a number of kernel objects (e.g., 20% in Linux 2.6.32) with an N-to-one mapping. We plan to address this issue in our future work as well.

Forth, there might be some execution contexts that are asynchronized. Consequently, we might miss the exact syscall context for the objects that are accessed in the asynchronized code. We have encountered a few cases in Linux (e.g., kernel worker threads, which are processed usually in the bottom-half of the interrupt context), and our current solution is to ignore tracking the context for these objects. Thus, an immediate future effort is to propose techniques that can also resolve the execution context of asynchronized execution code.

Finally, while we have demonstrated our techniques working for the Linux kernel, we would like to validate the generality of ARGOS with other kernels. We plan to extend our analysis to FreeBSD, since it is also open source and we can compare our result with its ground truth. Eventually, we would like to test our system with closed source OSes such as Microsoft Windows.

7 Related Work

Our work is closely related to data structure reverse engineering. More broadly, it is also related to virtual machine introspection and memory forensics. In this section, we compare ARGOS with the most closely related work—data structure reverse engineering.

Being an important component of a program, data structures play a significant role in many aspects of modern computing, such as software design and implementation, program analysis, program understanding [20], and computer security. However, once a program has been compiled, the definition of the data structure is gone. In the past decade, a considerate amount of research has been carried out to recover data structure knowledge from binary code, and they are all based on the same principle of "*from data use infer the data types*". In general, these existing approaches can be classified into two categories: static analysis based and dynamic analysis based.

Static Analysis. An early attempt of using static analysis to recover data structure is aggregate structure identification (ASI) [21]. Basically, it leverages the program's access patterns and the type information from well-known functions to recover the structural information about the data structures. While it focused on Cobol programs, its concepts can be applied to program binary code. By statically walking through the binary code, value set analysis (VSA) [3,22] tracks the possible values of data objects, from which it can build the point-to relation among addresses (which can help with shape analysis), and also reason about the integer values an object can hold at each program point. Most recently, TIE [16] infers both the primitive types and aggregate types of the program's variables from its binary code using the instruction type sinks and a constraint solving approach.

Dynamic Analysis. Guo et al. [12] propose an algorithm for inferring the variables' abstract types by partitioning them into equivalence classes through a data flow based analysis. However, this approach requires the program to be compiled with debugging symbols. A number of protocol reverse engineering efforts have been developed (e.g., [5,25,17,7]) to infer the format of network protocol messages—essentially the data structure type information of network packets—from program execution. The key idea of these approaches is to monitor the execution of network programs and use the instruction access patterns (i.e., the data use) to infer the data structure layout and size.

Rather than focusing on the data structure of network packets, REWARDS [18] shows an algorithm that can resolve the program's internal data structures through type recovery and type unification. Howard [23] recovers data structures and arrays using pointer stride analysis. PointerScope [27] infers pointer and non-pointer types using a constrained type unification [8]. Also, Laika [6] uses a machine learning approach to identify data structures in a memory snapshot and cluster those of the same type, with the applications of using data structures as program signatures.

Compared to all these existing works, ARGOS is the first system that focuses on the semantic reverse engineering of data structures. Also, nearly all of the existing work focused on the reverse engineering of user level data structures, and ARGOS makes the first step towards reverse engineering of kernel data structures.

8 Conclusion

We have presented ARGOS, the first system that can automatically uncover the semantics of kernel objects from kernel execution traces. Similar to many other data structure reverse engineering systems, it is based on the very simple principle of data-use implying data-semantics. Specifically, starting from the system call and the exported kernel APIs, ARGOS automatically tracks the instruction execution and assigns a bit vector for each observed kernel object. The bit vector encodes which syscall accesses this object and how the object is accessed (e.g., whether the object is created, accessed, updated, or destroyed under the execution of this syscall), and from this we derive the meaning of the kernel object. The experimental results with Linux kernels show that ARGOS can effectively recognize the semantics for a number of kernel objects that are of security interest. We have applied ARGOS to recognize the internal kernel functions, and we show that with ARGOS we can build a more precise kernel event tracking system by hooking these internal functions.

Acknowledgement

We thank our shepherd William Robertson and other anonymous reviewers for their insightful comments. This research was partially supported by an AFOSR grant FA9550-14-1-0119, and an NSF grant 1453011. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and NSF.

References

1. Linux test project. https://github.com/linux-test-project.
2. QEMU: an open source processor emulator. *http://www.qemu.org/*.
3. BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *CC* (Mar. 2004).
4. CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and and Communications Security (CCS'09)* (Chicago, Illinois, USA, 2009), pp. 621–634.
5. CABALLERO, J., AND SONG, D. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)* (Alexandria, Virginia, USA, 2007), pp. 317–329.
6. 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.
7. CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUN-BRIZ, L. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (Alexandria, Virginia, USA, October 2008), pp. 391–402.
8. DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 1982), pp. 207–212.
9. DENG, Z., ZHANG, X., AND XU, D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New Orleans, Louisiana, 2013), ACSAC '13, pp. 289–298.
10. FU, Y., AND LIN, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of* 33^{rd} *IEEE Symposium on Security and Privacy* (May 2012).
11. GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)* (February 2003), pp. 38–53.

12. GUO, P. J., PERKINS, J. H., MCCAMANT, S., AND ERNST, M. D. Dynamic inference of abstract types. In *ISSTA* (July 2006), pp. 255–265.
13. HAY, B., AND NANCE, K. Forensics examination of volatile system data using virtual introspection. *SIGOPS Operating System Review 42* (April 2008), 74–82.
14. JOHNSON, N., CABALLERO, J., CHEN, K., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of* 32^{nd} *IEEE Symposium on Security and Privacy* (may 2011), pp. 347 –362.
15. JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Boston, MA, 2006), USENIX Association.
16. LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)* (San Diego, CA, February 2011).
17. LIN, Z., JIANG, X., XU, D., AND ZHANG, X. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, February 2008).
18. 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).
19. NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)* (San Diego, CA, February 2005).
20. 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, USA, 1997), ICSE '97, pp. 338–348.
21. 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.
22. 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.
23. SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)* (San Diego, CA, February 2011).
24. WALTERS, A. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.
25. WONDRACEK, G., MILANI, P., KRUEGEL, C., AND KIRDA, E. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, February 2008).
26. ZENG, J., FU, Y., AND LIN, Z. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments* (Istanbul, Turkey, March 2015), pp. 147–160.
27. ZHANG, M., PRAKASH, A., LI, X., LIANG, Z., AND YIN, H. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (San Diego, CA, February 2012).