Automatic Uncovering of Tap Points from Kernel Executions

Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin(✉)

The University of Texas at Dallas, 800 W. Campbell Rd, Richardson, TX 75080, USA
{jzeng,yangchun.fu,zhiqiang.lin}@utdallas.edu

Abstract. Automatic uncovering of tap points (i.e., places to deploy active monitoring) in an OS kernel is useful in many security applications such as virtual machine introspection, kernel malware detection, and kernel rootkit profiling. However, current practice to extract a tap point for an OS kernel is through either analyzing kernel source code or manually reverse engineering of kernel binary. This paper presents AutoTap, the first system that can automatically uncover the tap points directly from kernel binaries. Specifically, starting from the execution of system calls (i.e., the user level programing interface) and exported kernel APIs (i.e., the kernel module/driver development interface), AutoTap automatically tracks kernel objects, resolves their kernel execution context, and associates the accessed context with the objects, from which to derive the tap points based on how an object is accessed (e.g., whether the object is created, accessed, updated, traversed, or destroyed). The experimental results with a number of Linux kernels show that AutoTap is able to automatically uncover the tap points for many kernel objects, which would be very challenging to achieve with manual analysis. A case study of using the uncovered tap points shows that we can use them to build a robust hidden process detection tool at the hypervisor layer with very low overhead.

Keywords: Virtual machine introspection · Kernel function reverse engineering · Active kernel monitoring · (DKOM) rootkit detection

1 Introduction

A tap point [10] is an execution point where active monitoring can be performed. Uncovering tap points inside an OS kernel is important to many security applications such as virtual machine introspection (VMI) [15], kernel malware detection [17], and kernel rootkit profiling [19,25]. For example, by tapping the internal execution of the creation and deletion of process descriptors, it can enable a VMI tool to track the active running processes [4]. Prior systems mainly hook the execution of the public exported APIs (e.g., system calls such as `fork` in Linux) to track the kernel object creation (e.g., `task_struct`). However, attackers can actually use some of the internal functions to bypass the check and create

F. Monrose et al. (Eds.): RAID 2016, LNCS 9854, pp. 49–70, 2016.
DOI: 10.1007/978-3-319-45719-2_3

the "hidden" objects. Therefore, it would be very useful if we can automatically identify these internal tap points and hook them for the detection.

Unfortunately, the large code base of an OS kernel makes the uncovering of tap points non-trivial. Specifically, an OS kernel tends to have tens of thousands of functions managing tens of thousands of kernel objects. Meanwhile, it has very complicated control flow due to the asynchronized events such as interrupts and exceptions. Finding which execution point can be tapped is daunting at binary level. In light of this, current practice is to merely rely on human beings to *manually* inspect kernel source code (if it is available), or reverse engineer the kernel binary to identify the tap points.

To advance the state-of-the-art, we present AutoTap, a system for AUTOmatic uncovering of TAP points directly from kernel binary. We focus on the tap points that are related to kernel objects since kernel malware often manipulates them. In particular, based on how an object is accessed, we classify the tap points into *creation, initialization, read, write, traversal,* and *destroy.* By observing which execution point is responsible for these object accesses, we derive the corresponding tap points.

The reason to derive tap points by associating kernel objects with the corresponding execution context is because different kernel objects are usually accessed in different kernel execution context. The context entails not only the instruction level access such as read or write to a particular field of an object, but also the calling context such as the current function, its call-chain, and its system call (syscall for brevity henceforth) if the execution is within a particular syscall, or other contexts such as interrupts. As to-be-demonstrated in this paper, such knowledge, along with the meaning of the available kernel data structures, is sufficient to derive the tap points for active kernel monitoring.

Having the capability of uncovering the tap points, AutoTap will be valuable in many security applications. One use case is we can apply AutoTap to detect the hidden kernel objects by tapping the internal kernel object access functions. Meanwhile, we can also use AutoTap to reverse engineer the semantics of kernel functions. For instance, with AutoTap we can now pinpoint the function that creates, deletes, initializes, updates, and traverses kernel objects. In addition, we can also identify common functions that operate with many different type of objects, which will be particularly useful to uncover the meanings of kernel functions especially for closed source OS.

In summary, we make the following contributions:

- We present AutoTap, the first system that is able to automatically uncover the tap points for introspection directly from kernel executions.
- We introduce a novel approach to classify the complicated kernel execution context into a hierarchical structure, from which to infer function semantics and derive tap points based on how an object is accessed.
- We have built a proof-of-concept prototype of AutoTap. Our evaluation results with 10 recent Linux kernels show that our system can directly recognize a large number of tap points for the observed kernel objects.
- We also show how to use our uncovered tap points to build a general hidden process detection tool, which has very small overhead.

2 System Overview

Since the goal of AutoTap is to uncover the tap points for the introspection of kernel object, we have to first track the kernel objects and their types. However, at binary code level, there is no type information associated with each object and we have to first recover them. Fortunately, our prior system Argos [30] has addressed this problem. Argos is a type inference system for OS kernels, and it is able to track the kernel object, assign syntactic types, and even point out the semantics for a limited number of important kernel objects but not all of them. Therefore, AutoTap has reused several components from Argos and also extended them for kernel object type inference.

Having inferred the types of kernel objects, we have to infer the tap points of our interest. A tap point is usually an instruction address (e.g., a function entry address) where active monitoring can be performed. Since tap points uncovering is essentially a reverse engineering problem, we have to start from known knowledge to infer the unknown one [10]. With respect to an OS kernel, the well-known knowledge would include its syscall interface (for user level programs), and all of its kernel module development interface (for kernel driver programs). Therefore, one of the key challenges would be how to leverage these knowledge to systematically infer the meaning of the accessed functions, from which to derive the corresponding tap points.

Key Insights. After analyzing how a typical OS kernel is designed and executed and also based on the experience from our prior systems including Argos and REWARDS [22], we have obtained the following insights to address the above challenges:

– **From data access to infer function semantics.** A program (with no exception to OS kernel) is composed of code and data, where code defines how to update data and data keeps the updated state. While there are a large number of kernel functions, from low level data access perspective, we can classify them into a number of primitive accesses including the basic data *read* and *write* according to how an instruction accesses it. We can also capture their lifetime based on their *allocation* and *deallocation* especially for heap and stack data. We can even differentiate further from the first time *write* (i.e., *initialization*) to the subsequent *write* according to the memory write operations. We can also conclude a piece of code is a *traversal* function if we observe it performs memory dereferences to reach other objects (either with the same type or different types).

– **From hardware level events to infer function semantics.** In addition to observing the instruction level data access behavior, we can also observe the hardware level events such as interrupts and exceptions to infer the function semantics. For example, if a function is always executed in a timer interrupt, we can conclude it is likely a periodic function (e.g., `schedule` function); if it is executed inside a keyboard response interrupt, we can conclude it is a keystroke interrupt handler.

- **From syscall level events to infer function semantics.** Another category of useful information is the system call events. If a function is executed inside `fork`, we know this function is likely kernel object creation related; if it is inside socket `send`, we know it must be network communication related. Meanwhile, we also know a `fork` syscall must create kernel objects such as `task_struct`, and a `send` syscall must access a `socket` object.
- **Inferring the semantics of objects from kernel APIs.** While kernel has a large number of kernel objects, not all of them are of attackers' interest. Consequently, we have to identify the type of the kernel objects such that we can pinpoint the tap points of our interest. To this end, we can leverage the types of the parameters and return values of kernel APIs, the public exported knowledge used when developing kernel modules, to resolve the object types (such as whether the object is a `task_struct`). Meanwhile, kernel developers often have access to a number of kernel header files (otherwise their modules may not be compiled). By combining the types resolved from the API arguments and the data structure definitions from the open header files, we can reach and resolve more kernel data structures.

Scope, Assumptions, and Threat Model. To make our discussion more focused, we target OS kernels executed atop a 32-bit ×86 architecture. To validate our experimental results with the ground truth, we use the open source Linux kernels as the testing software[1]. Regarding the scope of the tap points, we focus on those that are related to dynamically allocated kernel heap objects.

As alluded earlier, we assume the knowledge of kernel APIs, e.g., the kernel object allocation function (e.g., `kmalloc`, and `kfree`) such that AutoTap can hook and track the kernel object creation and deletion, and the types of the arguments of the kernel APIs, which will be used to resolve the kernel object types. Meanwhile, we assume the access of the header files related to kernel module development (this is also true for Microsoft Windows), and the data structure defined in the header files will also be used to type more kernel objects. AutoTap aims to discover the tap points for introspection in the existing kernel legal binary code. If there is any injected code, AutoTap cannot tap their executions.

Overview. We design AutoTap by using a binary code translation based virtual machine monitor (VMM) Pemu [29], which extends Qemu [2] with an enhanced dynamic binary instrumentation capability. There are three key components inside AutoTap: *kernel object tracking*, *object access resolution*, and *tap points uncovering*; they work in two phases: an online tracing phase, and an offline analysis phase (Fig. 1).

[1] Note that even though the kernel source code is open, it is still tedious to derive the tap points manually, and a systematic approach such as AutoTap is needed.

In the online phase, starting from the kernel object creation, *kernel object tracking* tracks the dynamically created kernel objects, their sizes, and their propagations and indexes them based on the calling context of the object creation for the monitoring. Whenever there is an access to these monitored objects, *object access resolution* captures the current execution context, resolves the types of the arguments, resolves the current access (e.g., whether it is a read, write, initialization, allocation, or deallocation), and keeps a record of the current object access with the captured execution context if this record has not been stored yet in the memory. Once we have finished the online tracing, we then dump the memory meta-data into a log file, and our *tap points uncovering* will analyze the log file to eventually derive the tap points. Next, we present the detailed design and implementation of these components.

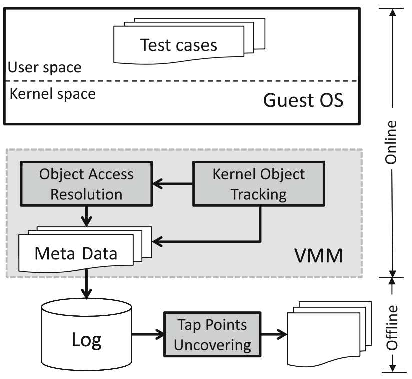

Fig. 1. An overview of AutoTap.

3 Design and Implementation

3.1 Kernel Object Tracking

As the focus of AutoTap is to extract the tap points related to the dynamically allocated kernel objects, we have to first (i) track their life time, (ii) assign a unique type to each object, (iii) track the object propagation and its size such that we know to which object the address belongs when given a virtual address, and (iv) resolve the semantic types of kernel objects. Since our prior system Argos also need to perform these tasks for its type inference, we reused a lot of code base to handle kernel object tracking. However, there are still some differences between AutoTap and Argos on (ii) how we assign syntactic type to each object and (iv) how we resolve the semantic type of object. Next, we just describe these differences. More details on (i) how we track object life time and (iii) resolve the object size can be found in Argos [30].

Assigning a Syntactic Type to Each Object. In general, there are two standard approaches to convert dynamic object instances into syntactic forms: (1) using the callsite address of `kmalloc`, denoted as $PC_{kmalloc}$ to represent the syntactic object type, or (2) using the callsite-chain of `kmalloc`, denoted as $CC_{kmalloc}$ to represent the syntactic object type. The first approach is intuitive but it cannot capture the case where `kmalloc` is wrapped. While the second approach can capture all the distinctive object allocation, it may over classify the object types since the same type can be allocated in different calling context.

Argos used the first approach since it aims to uncover the general types (context-insensitive). In AutoTap, we adopt the second approach because we aim to identify the execution point for the tapping, and these points are usually context sensitive. For instance, a string is a general type but when it is used in different contexts (e.g., to represent a machine name or a process name), it means different type of strings and we may just need to tap a particular type of string instead of all strings (that is why sometimes we have context-sensitive tap points). Therefore, we use $CC_{kmalloc}$ to denote the syntactic type for each dynamic allocated object. The semantic meaning of $CC_{kmalloc}$ will be resolved later. Also, we use a calling context encoding technique [27] to encode $CC_{kmalloc}$ with an integer $E(CC_{kmalloc})$, and store this integer and its corresponding type with a hash table we call HT_{type} for easier lookup and decoding.

Resolving the Semantic Type of Object. The syntactic type ($CC_{kmalloc}$) assigned to each object is only used to differentiate objects, and it does not tell the semantics (i.e., the meaning) of the object. Since the tap points we aim to uncover are associated to each specific kernel object (e.g., `task_struct`), we need to resolve their semantic types. While Argos can recognize semantics for a number of kernel objects if there are unique rules to derive their meanings under certain syscall context, it cannot recognize all kernel objects. Therefore, we use a different approach, which is inspired by our another prior system REWARDS [22], a user level data structure type inference system. In particular, REWARDS infers the semantics of data structures through the use of well-known semantic type information from the argument and return value of system call and user level APIs. We adopt the RWEARDS approach to infer the kernel object semantic types from public known kernel APIs. However, not all objects can be typed from the argument and return value of these APIs, and therefore we also leverage the object types defined in the header files for kernel module development and track object point-to relations to infer more object types. To capture the point-to relation between objects, we use the same taint analysis approach as in Argos.

Summary. Our *kernel object tracking* will track the life time of the dynamically allocated objects with a red-black tree we call $\mathrm{RB}_{instance}$ tree that is used to store $<v, s, T_i, E(CC_{kmalloc})>$, which is indexed by v, where v is the starting address, s is the corresponding size, T_i is the taint tag for O_i, and $E(CC_{kmalloc})$ is the encoded syntactic type of the allocated object. Also, it will maintain a hash table we call HT_{type} that uses $E(CC_{kmalloc})$ as the index key. This HT_{type} stores the decoded callsite chain, the resolved semantic type of the objects based on kernel APIs and available header files, as well as the captured point-to relations between them. Also, the resolved access context to each field of a particular type (described next) is also stored in our HT_{type}.

3.2 Object Access Resolution

Once we have captured each kernel object and its (field) propagations, the next step is to resolve the execution context when an instruction is accessing our monitored object. Note that the execution context *captures how and when a piece of data gets accessed.* In general, when a piece of data gets accessed, under dynamic binary code instrumentation based VMM, what we can observe includes: (i) which instruction is accessing the data, (ii) through what kind of access (read, or write). However, such information is still at too low level, and what we want is the high level semantic information that includes (i) which execution context (e.g., syscall, interrupt, kernel thread) is accessing the object and under what kind of calling context, and (ii) what the concrete operation is with respect to the accessed object (e.g., create, read, write, initialize, allocation, deallocation). Therefore, we have to bridge this gap.

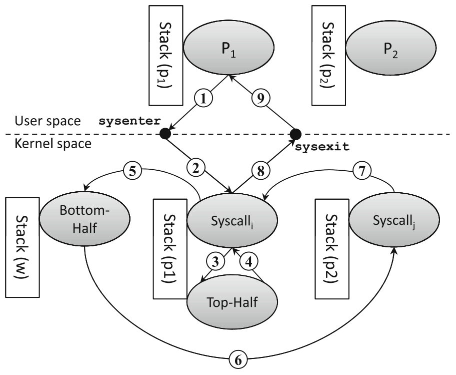

Fig. 2. An illustration of the three top level kernel execution contexts.

A kernel execution context in fact has a hierarchical structure and it can be classified into three layers. From top to bottom, there are syscall level context, function call level context, and instruction level context. In the following, we describe how we resolve these contexts and associate them to the accessed kernel objects.

Resolving Top Level Execution Context. When a given kernel object gets accessed, we need to determine under which highest level execution context it is accessed. As shown in Fig. 2, there are *three kinds of disjoint highest level execution contexts*:

– **(I) Syscall execution context.** When a user level program requests a kernel service, it has to invoke the syscalls. When a syscall gets executed, kernel control flow will start from the entry point of the syscall, and continue its execution until this syscall finishes. There is always a corresponding kernel stack for each process that tracks the return address of the functions called by this syscall. Therefore, we have to first identify to which process the current syscall belongs, and identify the entry point and exit point of this syscall.
  In ×86, the entry point and exit point of a syscall for Linux platform can be easily captured by monitoring the syscall enter and exit instructions (e.g., `sysenter`, `sysexit`, `int 0x80`, `iret`). To identify a process context, we use the base address of kernel stack pointer, i.e., the 19 most significant bits of the kernel `esp`, denoted MSB19(`esp`), since kernel stack is unique to each process or kernel thread, as what we have done in Argos.
  Therefore, as shown in Fig. 2, when an instruction is executed between the syscall entry (Control Flow Transition ①, *CFT*① for brevity) and exit point

(*CFT*⑧), if it is not executed in an interrupt handler's context (discussed below), and if the context belongs to the running process, then it is classified the syscall execution context. We will resolve the corresponding syscall 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, which is computed by monitoring the memory write to the kernel `esp`. We also use another RB-tree, and we call it RB_{sys} tree to dynamically keep the MSB19(`esp`) and the `eax` that is the syscall number, for each process such that we can quickly retrieve the syscall number when given a kernel `esp` if the execution is executed inside a syscall.

- **(II) Top-half of an interrupt handler execution context.** While most of the time kernel is executed under certain syscall context for a particular process, there are other asynchronous kernel events driven by the interrupts and exceptions, and they can occur at any time during the syscall execution. To respond them, modern OS such as Linux kernel usually splits the interrupt handlers into top-half that requires an immediate response and bottom-half that can be processed later [6].
  As illustrated in Fig. 2, top half of an interrupt can occur at anytime during a syscall execution (e.g., when a time slice is over, a key is stroke, or a packet is arrived). It starts from a hardware event (*CFT*③ which can be monitored by our VMM), and ends with an `iret` instruction (*CFT*④). The execution of a top-half is often very short, and it can use the kernel stack of the interrupted process to store the return address if there is any function call, or use a dedicated stack for this particular interrupt depending on how the interrupt handler is implemented. Meanwhile, an interrupt execution can be nested. Thus, we have to capture the pair of *CFT*③ and *CFT*④. This can be tracked by using a stack-like data structure. Through such, the top half of an interrupt handler can be precisely identified.
- **(III) Bottom-half of an interrupt handler execution context, or kernel thread execution.** When the response for an interrupt takes much longer time, kernel often leaves such an expensive execution to dedicated kernel threads (to execute the bottom half of an interrupt handler) such as `pdflush`, `ksoftirqd`. Therefore, *there must be a context switch event*, which can be observed by the kernel stack exchange. Note that *CFT*⑤, *CFT*⑥, and *CFT*⑦ all denotes the context switch event because of the stack exchange. In other words, as illustrated in Fig. 2, we can actually uniformly treat them as the syscall context of user level processes with the only difference that they do not have a syscall entry and syscall exit point.

Resolving Middle Level Execution Context. Having identified the highest level execution context, we also need to identify the middle level execution context at a function call level that includes which function is executing the current instruction and the callers of this function. Naturally it leads us to identify the function call chain. While we can get the call chain by traversing the stack frame pointer, it requires kernel to be compiled with this information. To make AutoTap more general, we instrument `call`/`ret` instruction and use a shadow stack to track the callsite chain. Based on the above three high level disjoint execution contexts, we maintain the following three kinds of shadow stacks (SS):

- **(I) Syscall SS.** When a syscall execution (say s_i) starts, we will create a corresponding SS(s_i). Then whenever there is a function call under the execution of s_i, we additionally push a tuple <f_entry_addr, f_return_addr, stack_ret_offset> into the corresponding SS(s_i), and whenever there is a `ret` executed under this syscall context, we additionally pop the tuple whose f_return_addr matches the return address from the top of SS(s_i). Note that without this matching check, there could exist cases that call and return are not strictly paired. Also, the `push/ret` of the return address when calling f will still use the original stack. The reason of tracking the stack_ret_offset in the original stack is for quickly retrieving of the entire calling context for context-sensitive tap points, when given just a kernel stack without instrumenting any call instructions. Then at any moment, the callsite chain for the current syscall context can be created by retrieving the value of f_return_addr in the corresponding kernel stack based on the location specified by stack_ret_offset.
- **(II) Top-half SS.** When a top half of an interrupt handler for interrupt i (say i_i) is executed, we also create a corresponding SS(i_i) to track the call chain for this interrupt context. When the interrupt returns (observed by `iret`), we clear this shadow stack. At anytime during the execution of this interrupt, we similarly build its callsite chain from SS(i_i) as what we do in the syscall context.
- **(III) Kernel Thread SS.** If the execution is neither in the syscall context, nor top half of the interrupt handler, then it must be in kernel thread execution context (or bottom half of an interrupt), say t_i. Similarly, we will create a corresponding SS(t_i) for each of this context. As such, we can retrieve the callsite chain when a kernel object is accessed under this context.

It should be noted that at runtime there can be multiple instances of each of these SS, because there can be multiple processes, interrupts, and kernel threads. We will extract the callsite chain from the corresponding one based on the value of MSB19(`esp`).

Resolving Low Level Execution Context. Once we have resolved all these high level execution contexts, our final step is to resolve the low level context (e.g., read/write) of how an object is accessed and keep a record in the in-memory meta-data (i.e., our HT_{type}). Currently, we focus on seven categories of accesses as presented in Table 1.

Specifically, whenever there is an access to the monitored kernel object O_i (including its k-th field F_k and the propagations), we will insert an entry if this has not been inserted to the field F_k's access list that is stored in HT_{type}, which is indexed by the encoded syntactic type of O_i (i.e., $E(CC_{kmalloc})$), and this entry consists of $<AT, EX>$ where AT denotes the access types of the seven different categories, and EX denotes the current execution context.

Table 1. Resolved access types based on the behavior.

Category	Behavior
Creation (O_i)	O_i is created by calling `kmalloc`
Deletion (O_i)	O_i is freed by calling `kfree`
Read (O_i, F_j)	A memory read field F_j of O_i
Traversal (O_i, F_j)	Read (O_i, F_j) $\wedge$ $F_j \in$ pointer field
Write (O_i, F_j)	A memory write to field j of O_i
Initialization (O_i, F_j)	Write (O_i, F_j) $\wedge$ first time write to F_j
Others	Other contexts, e.g., periodical access

To save both memory and disk space of our meta-data, we also encode EX. Basically, EX is composed with (1) the low level access behavior that includes the program counter (PC) of Read, Traversal, Write, Initialization of (O_i, F_k), or the entry address of `kmalloc` or `kfree` if it is object creation/deletion, as well as the encoding of these accesses; (2) middle level callsite chain and the corresponding offset in the running kernel stack to locate each function's return address; and (3) the top level context that is either a syscall number, or an interrupt number, or the value of MSB19(`esp`) of kernel thread. We also encode EX with an integer and use a hash table to store the mapping between the integer and the concrete execution context. Our *tap points uncovering* will scan the dumped meta-data to eventually uncover the tap points.

3.3 Tap Points Uncovering

Once collected the record describing how a particular type of monitored kernel object is accessed, the final step of AutoTap is to perform an offline analysis to further derive the tap points for each type of kernel object. At a high level, for a given syntactic type of a kernel object, we traverse our memory-dumped HT$_{type}$ and locate its field access context <AT, EX>. For each EX, we rebuild a context-chain according to our encoding. The top of the chain is the highest level execution context (i.e., syscall, interrupt, or kernel thread), followed by the callsite chain. Examples of such context-chains are illustrated in Fig. 3. After having the context-chain, we are then ready to extract the tap points.

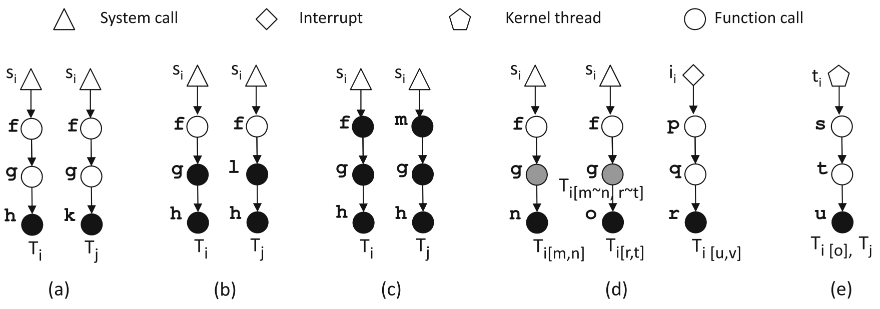

Fig. 3. Enumerated and simplified cases for tap points uncovering. Note that s_i denotes i_{th} syscall, i_i denotes i_{th} interrupt, t_i denotes i_{th} kernel thread, f, g, h etc. all denotes function calls, T_i represents the syntactic type, and $T_{i[m]}$ denotes the field m of T_i.

Introspection Related Tap Points. Among all the tap points, those related to object creation, deletion, traversal, and field read are of particular interest to introspection, especially for the detection of hidden kernel objects. In the following, we present how we uncover these tap points:

– **Object Creation and Deletion.** Given a specific syntactic type T_i (note that syntactic type is used to find the tap points, and semantic type is used to pinpoint the one we want) for a kernel object, we scan the context-chain, if the leaf node of the chain creates/deletes a kernel object with the matched type, then the tap points in this context chain will be included in the result. Ideally, if the leaf node is unique among all the types, we can directly output the PC that calls the leaf function as the corresponding tap points for this type. However, these functions might also create other types of object. Therefore, we will scan the context-chain again, and compare with other types to produce the final result.
  Specifically, there are at most three cases for the creation and deletion related tap points. One is the leaf node that is unique among all observed types (Fig. 3(a)), and as discussed we directly output the call-site PC of the leaf function as the tap points (function h and function k in this case) and these tap points are context-insensitive. Otherwise, we scan further and compare their parent functions (Fig. 3(b) and (c)). If they differ at their closest parent function, then we use the call-chain from the diffed parent function to the leaf node (Fig. 3(b)) and use the chained call-site PC as the tap points and these tap points are context-sensitive; otherwise we will scan until we reach their root node, and in this case we will use the entire context chain (Fig. 3(c)). Recall that there must exist a unique callchain for each syntactic object (Sect. 3.1). Therefore, we will not have a case in which we cannot find the unique context chain even though we have reached the root.
– **Object Traversal.** The tap points for object traversal are critical for introspection, especially if we aim to identify the hidden objects. To identify such tap points, we scan the context chain: if we observe there is a pointer field read from object O_i to reach object O_j, we conclude there is an object traversal in the observed function with the tap point of the PC that performs the read operation. If this PC only accesses this particular type of object, we just use this PC as the tap points; otherwise, we will use the call-chain as what we do in object creation/deletion tap points discovery. Also, we can identify recursive type traversal if both O_i and O_j share the same type, otherwise it will be a non-recursive traversal.
– **Object Field Read.** Pointer field read can allow us to identify the object traversal tap points. Non pointer field can also lead to certain interesting tap points. Similarly to how we identify object traversal tap points where we focus on the pointer field, we will also derive all the non pointer field read tap points.

Other Tap Points. In addition, there are also other types of tap points, such as object field initialization and object field (hot) write. Though these tap points may not be directly used in introspection, they could be useful for kernel function reverse engineering in general. AutoTap does support identify these tap points. For instance, it becomes straightforward to identify the initialization point (the first time memory write). The only issue is there may not exist a centralized function that initializes all the field of an object. For example, as shown in Fig. 3(d), the leaf node may just initialize partial fields of an object. Therefore, we need to hoist the field initialization information to their parent functions. Such hoist operation is a recursive procedure and we will use the lowest parent function that cannot expand the scope further of the fields of T_i as the initialization tap points for the observed field. We are also interested in several other particular interesting types of tap points, such as the periodic functions that are executed in the timer interrupts. We will demonstrate how to use these tap points in Sect. 5.

4 Evaluation

We have implemented AutoTap. The online analysis component is built atop Pemu [29] by reusing a large amount of code base from Argos [30], and the offline component is built using python. In this section, we present our evaluation result.

Experiment Setup. The input to AutoTap is the kernel API specification, the available kernel data structure definitions for kernel module developers, and the test cases to run the kernel. We acquired kernel API specification, namely, function name, the type of its arguments and return values from `/lib/modules/KERNEL_VERSION/build`. We extracted the kernel data structure definitions from the available kernel header files. In order to intercept the kernel APIs for object tracking and semantic type inference, we identified their function entry addresses from `/proc/kallsyms`.

To run the kernel, we used the test cases from the Linux-test-project [1], as what we have done in FpCk [14]. We took 10 recent released Linux kernels, presented in the first column of Table 2, as the guest OS for the test, and executed them inside our VMM. The testing host OS runs `ubuntu-12.04` with kernel `3.5.0-51-generic`. The evaluation was performed on a machine with a 64-bit Intel Core i-7 CPU with 8 GB physical memory.

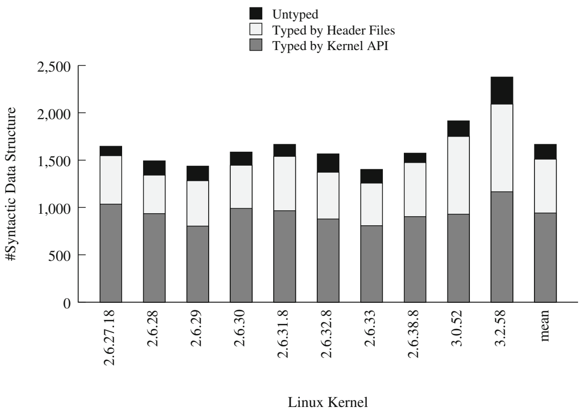

Fig. 4. Type resolution result for each kernel

To identify a tap point for a particular type of object, AutoTap first derives all the tap points for each syntactic type, and then us0es the resolved semantic type (e.g., `task_struct`) associated with the syntactic type to eventually pinpoint the tap points of introspection interest. Therefore, we first present the result regarding how AutoTap performed to identify the tap points for the syntactic type, and then the tap points for the semantic type.

Result for Syntactic Types. We first report how our *kernel object tracking* component performed in Fig. 4. As shown in this figure, our *kernel object tracking* component identifies on average 1.8 thousand unique syntactic types. We can see about 57 % of them can be semantically typed by using the kernel APIs. With the public open kernel module development header files, it can type additionally 35 % of them. In other words, close to 90 % of the data structures can be semantically typed.

Next, we report how our second and third components performed in Table 2. Specifically, the result of our *object access resolution* is reported from the 2nd column to the 7th column. The number of the top level context, namely syscall context, is reported in the 2nd column, interrupt in the 3th column, and kernel thread in the 4th column. We can notice that on average, AutoTap observed 219 system call contexts, 7 interrupt/exception contexts (e.g., `page fault`, timer, keyboard, `device-not-available`), and 29 kernel thread contexts. Regarding the middle level context, we report the total number of function call-site chain in the $|FC|$ column, and there are 104,971 unique call-site chains associated with these traced types. Finally, for the lowest level context, we report the total number of field read tap points in $|PC_R|$ and write tap points in $|PC_W|$ columns. We can notice that there is a significant large number of the unique field read/write access contexts. If we perform manual analysis, it is very challenging to systematically identify them all.

Finally, we report the statistics of the tap points uncovered for the introspection in the rest columns of Table 2. In total, we report five categories of introspection related to tap points: object creation, object deletion, object recursive type traversal ($R_{Traversal}$), object non-recursive type traversal ($N_{Traversal}$), and object field read (F_{Read}). For each category, we report the number of the tap points that are context-insensitive (i.e., we can directly use the corresponding PC as the tap points) in column $|PC|$, and context-sensitive (i.e., we need to inspect the call-chain in the corresponding stack frame when the PC is executed)

Table 2. Overall result of tap points uncovered for each tested kernel.

Kernel	Object Access Resolution						Tap Points Uncovered									
	$\|Sys\|$	$\|Int\|$	$\|Thd\|$	$\|FC\|$	$\|PC_R\|$	$\|PC_W\|$	$Creation$		$Deletion$		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
							$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$
2.6.27.18	219	7	23	77634	308643	136729	0	1646	47	1507	89	2408	1402	21585	4209	61632
2.6.28	218	7	22	73285	313488	134027	0	1492	61	1452	89	2313	1435	18460	4235	54706
2.6.29	216	7	29	69547	313442	132004	0	1436	59	1485	90	2375	1515	18251	4102	56866
2.6.30	217	7	28	40457	319834	136593	0	1585	62	1506	97	2341	1598	20303	4367	62927
2.6.31.8	217	7	28	74121	346884	147573	0	1666	66	1560	97	2497	1482	21679	4159	74504
2.6.32.8	218	7	31	92690	450004	194353	0	1566	54	1365	93	2322	1500	18192	3943	62115
2.6.33	217	7	31	85544	412563	176407	0	1402	64	1274	94	2208	1221	14084	4082	65531
2.6.38.8	217	7	33	91438	422170	185327	0	1573	56	1293	97	2479	1541	18881	3838	62361
3.0.52	222	7	36	205984	797643	238132	0	1915	68	1768	113	2695	1695	20538	4445	66432
3.2.58	227	7	35	239018	898387	270936	0	2377	71	2085	109	3967	1739	27619	4373	89204
Average	219	7	29	104971	458305	175207	0	1654	62	1545	97	2560	1672	19959	4175	65628

in column $|FC|$. We can notice that there are many context sensitive tap points because different syntactic types (which is from the same semantic type) use the same PC for the allocation, but in different calling context. We can also notice some tap points can be used to delete different type of object (e.g., in Linux kernel 2.6.32.8, there are 1566 syntactic types allocated, but it only requires 1365 deletion tap points), and there are too many object traversal tap points, which proves it will be extremely difficult to identify them with just purely manual analysis. Regarding how to use the derived tap points, we present a case study in Sect. 5.

Result for Semantic Types. As shown in Table 2, there are too many tap points. To really use them for introspection, we have to select the ones of our interest. Therefore, we have to get the tap points based on the semantic types. We take Linux-2.6.32.8 as an example, and describe in greater details how this is achieved.

For Linux-2.6.32.8, as our syntactic type is an over-split of the semantic types (i.e., multiple syntactic types can correspond to just one semantic type), our technique eventually resolved the semantic types of 87.6 % (1372/1566) of the syntactic types. Once we have resolved the semantic types, we have to iterate our tap points uncovering again for each semantic types using the same algorithm described in Sect. 3.3.

Take `task_struct` as an example, before applying the semantic types, we acquired 6 different syntactic types of `task_struct`, namely, each of these is created in a different call-chain. The (64-bit) integer encoding of these syntactic types are presented in the first column of Table 3. For object creation, each of these syntactic types has a context-sensitive tap point, and none of them is context-insensitive; similar result also applies to object deletion. For recursive traversal, we observed the 3rd syntactic type of `task_struct` has a heavy recursive traversal. Compared with other syntactic type, this one has many more `task_struct` instances. For non recursive type traversal, each syntactic type has a lot of context-sensitive pointer read. Finally, for the object field (i.e., non-pointer) read, we can notice most of their tap points are context sensitive.

Table 3. Tap points statistics for 6 different syntactic types of `task_struct`.

Syntactic type	$Creation$		$Deletion$		$R_{Traversal}$		$N_{Traversal}$		F_{READ}	
	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$
4dd23b5e689e2ad7	0	1	0	1	0	9	3	102	1	299
536881ec388d6516	0	1	0	1	1	7	20	225	36	420
7554a8d7acf81704	0	1	0	1	41	131	403	402	435	563
8649536d24938b96	0	1	0	1	0	0	0	304	1	437
9ac37673946479aa	0	1	0	1	0	30	0	136	14	318
9d41a458fa47a47b	0	1	0	1	0	0	2	289	0	448

Table 4. The statistics for the uncovered tap points for the observed semantic types of `linux-2.6.32.8` in slab/slub allocators

Category	**Semantic Type**	**#Syntactic Type**	*Creation*		*Deletion*		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
			$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$
Process	task_struct	6	1	0	1	0	98	93	725	6	1024	24
	pid	6	1	0	1	0	2	1	15	3	50	1
	task_delay_info	6	1	0	1	0	0	0	0	0	24	4
	task_xstate	7	2	0	1	0	0	0	0	0	38	1
	taskstats	2	1	0	1	0	0	0	0	0	27	0
Memory	anon_vma	7	1	0	1	0	0	0	5	1	8	1
	mm_struct	4	2	0	1	0	0	0	21	8	235	32
	vm_area_struct	44	7	0	2	0	84	94	113	1	395	1
Network	TCP	3	0	1	0	1	7	0	74	8	1023	137
	UDP	2	0	1	0	1	0	0	0	0	0	84
	UNIX	4	0	1	0	1	8	0	29	4	118	36
	neighbour	7	1	0	1	0	2	0	4	0	113	15
	inet_peer	1	1	0	1	0	0	0	0	0	23	1
	rtable	7	1	0	1	0	0	0	11	0	155	3
	nsproxy	1	1	0	1	0	0	0	1	0	6	0
	request_sock_TCP	2	1	0	1	0	0	0	1	0	70	8
	skbuff_fclone	7	0	1	0	1	0	0	76	78	89	161
	skbuff_head	53	1	1	0	1	1	0	152	78	148	161
	sock_alloc	4	1	0	1	0	0	4	64	2	59	34
File	bio-0	94	0	1	0	1	3	0	18	0	123	30
	biovec-16	5	0	1	0	1	0	0	0	0	0	26
	biovec-64	4	0	1	0	1	0	0	0	0	1	30
	io_context	17	1	0	1	0	0	0	7	2	15	7
	request	60	0	1	0	1	13	99	22	0	164	2
	dentry	85	1	0	1	0	80	4	321	4	197	10
	ext2_inode_info	4	1	0	1	0	6	17	74	12	136	262
	ext3_inode_info	21	1	0	1	0	6	19	38	35	580	348
	fasync_struct	1	1	0	1	0	0	0	1	0	1	1
	file_lock	10	1	0	1	0	11	6	17	0	113	3
	files_struct	4	1	0	1	0	0	3	25	10	41	41
	file	33	1	0	1	0	4	5	227	7	352	4
	fs_struct	4	1	0	1	0	0	0	9	2	44	3
	inode	5	1	0	1	0	2	5	5	8	15	113
	journal_handle	124	1	0	1	0	0	0	28	0	25	0
	journal_head	82	1	0	1	0	19	0	66	0	50	0
	proc_inode	9	1	0	1	0	0	0	6	3	33	95
	sysfs_dirent	36	1	0	1	0	12	0	7	0	31	0
	vfsmount	4	1	0	1	0	31	0	21	8	63	3
IPC	mqueue_inode_info	1	1	0	1	0	0	0	15	2	37	49
	shmem_inode_info	8	1	0	1	0	0	4	0	16	107	194
Signal	fsnotify_event	19	1	0	1	0	1	0	8	2	24	2
	inotify_event_private_data	19	2	0	1	0	0	0	3	0	2	0
	inotify_inode_mark_entry	1	1	0	1	0	1	0	7	1	25	1
	sighand_struct	6	1	0	1	0	0	0	0	0	66	4
	signal_struct	6	1	0	1	0	0	12	11	4	265	36
	sigqueue	17	1	0	1	0	4	2	8	2	8	0
Security	cred	41	2	0	1	0	0	3	28	3	352	1
	key	4	1	0	1	0	0	10	4	0	53	3
Other	buffer_head	61	1	0	1	0	20	0	21	0	423	0
	cfq_io_context	17	1	0	1	0	2	0	15	3	39	1
	cfq_queue	15	1	0	1	0	0	0	17	5	106	1
	idr_layer	12	1	0	3	0	5	5	1	3	19	3
	names_cache	58	2	0	3	0	0	0	0	0	16	10
	k_itimers	1	1	0	1	0	1	0	12	0	24	24
	radix_tree_node	56	1	0	1	0	10	3	2	3	22	9
	jbd_revoke_record_s	14	1	0	1	0	1	0	0	0	7	0

After we apply the resolved semantic type to each syntactic type and re-execute our tap points uncovering, many of the context-sensitive tap points become context-insensitive. For instance, for `task_struct`, as illustrated in the first row of Table 4, these 6 syntactic types get actually merged into one, and we then can directly use the PC for object creation and deletion without inspecting their call-stack. Due to space reason, we report the tap points uncovering statistics for some of the resolved semantic types in Table 4. In total, there are over 90 resolved semantics, and we only report 56 of them that are visible in the slab allocators.

Performance Result. Regarding the performance of AutoTap, for each tested kernel, our online analysis took around 12 h on average to finish the testing benchmark, and our offline analysis took just a few minutes to process the log files and produce the final tap points. The dumped log file size is around 500 MB (thanks to our encoding). The reason why our online analysis took so long is because we have one thousand test cases to execute and also we have to perform dynamic binary instrumentation to track object and field propagations in our instrumented VMM.

5 Security Application

In this section, we demonstrate how to use our tap points for a particular type of introspection application—hidden process detection. Typically when a system is compromised, it is often very common for attackers to hide the presence of their attack and also leave certain invisible services for future privileged access. To achieve this, one simple way is to keep running of a privileged process, and hide it from the system administrators through rootkit attacks.

At a high level, there are three different categories of rootkits for process hiding [18,25]. The first category directly modifies program binaries such as `ps`, `pslist`, etc. The second category hooks into the call path between a user application and the kernel by modifying system libraries (e.g., `glibc`), dynamic linker structures (`plt/got` table), system call tables, or corresponding operating system functions that report system status [28]. The third category manipulates kernel data structures using the so-called direct kernel object manipulation (DKOM) [12] attack, such as removing the process descriptor (e.g., `task_struct`) from the accounting list shown by `ps`.

Our Approach. Since AutoTap has extracted the tap points related to the `task _struct`, especially the creation/deletion and traversal tap points, it would enable the monitoring and detection of the hidden processes. One intuitive approach is to use the tap point that traverses all the elements in the accounting `task` list. However, we did not find such a tap point that iterates all the element of the `task` list. In fact, utility command such as `ps` will not traverse the accounting list to show all the running process, and instead it extracts the process list from the `/proc` file system [13].

While there are many traversal tap points for the `task_struct`, as shown in Table 3, there must be some traversal tap points executed by the `schedule` function. Note that `schedule` function is very easy to identify as it is always executed in the top half of the timer interrupt handler (though it can be called in various other places), and meanwhile there must be a stack exchange (a kernel `esp` write operation). Therefore, if we can identify the `task_struct` accessed by the `schedule` function, and if we can know to which `task_struct` instance the CPU switches, then we can identify the `task_struct` that is to-be-executed by the CPU.

However, we have to solve another challenge—how to identify the to-be-executed `task_struct` instance given that `schedule` function may access a number of other `task_struct` instances to pick up the next to-be-executed one (defined by the policy) for the execution. Fortunately, as we have noted, when performing a context switch, there must be a stack pointer exchange, and the new stack pointer must come from the to-be-executed process. Typically, this stack pointer is stored in `task_struct`. Therefore, by monitoring where the stack pointer comes from, we identify the to-be-executed `task_struct` instance. Recall that we have tracked all field (and its propagation) read, and we just need to identify this particular field.

More specifically, we found 123 Object Traversal tap points for `task_struct` in the context of `schedule` function. In particular, there are 76 recursive and 26 non-recursive traversal tap points. All of them are context insensitive. Part of the reason we believe is `schedule` function is very unique and other functions will not call it for other purposes other than scheduling. Among these 123 tap points, we know one of them must be of our interest since we aim to capture the `task_struct` traversal. Also, we found 121 `task_struct` Field Read, all of which are also context insensitive. By looking at these field read tap points, we found there is a particular field read tap point that uses the stack pointer (i.e., `0xc125e3b1:mov 0x254(%edi),%esp`). Interestingly, the base register `edi` here actually holds the address of the to-be-executed `task_struct`. Therefore, we actually do not need the traversal tap points and we just need to hook this tap point, because we can directly identify the to-be-executed process from `edi`.

From the above analysis, we can notice that with AutoTap, we have significantly reduced the search space of the instruction of our interest from tens of thousands (4,422 instructions in the context of `schedule` function in which a manual analysis has to analyze) to only a few hundred (123 object traversal, and 121 field read). With insight of how context switch is performed, we further reduce the search space to only a few instructions (it is `0xc125e3b1:mov 0x254(%edi),%esp` in our case). This is just one case we demonstrated for `task_struct`. Regarding many other kernel data structures, our system also applies even though we may have to consider certain data structure specific insight. For instance, if we want to detect hidden `socket`, we can use the insight that `socket` must be accessed at system call send/sendto/write or recv/recvfrom/read context.

The Detection Algorithm. We use a cross-view comparison approach that compares the CPU time execution from inside and outside to detect the hidden processes. Note that CPU time metric is the most reliable source (tamper-proof) for rootkit detection. In particular, to detect rootkit, we first get an inside view by running `ps` command, and an outside view by counting the CPU `TIME` for the running process. In particular, the inside view will show the running process `PID`, `TTY`, `TIME`, and `CMD`. Among them, `TIME` is very critical and it is very challenging (nearly impossible) for attacker to forge a value that will be equivalent to the one counted at the hypervisor introspection layer.

Table 5. Process hiding rootkits

Rootkits	Process hiding mechanism
ps_hide	Fake ps binary with process hiding function
libprocesshider	Override libc's readdir to hide process
LinuxFu	Hide the process by deleting its `task_struct` from task list

To count the executed time for a particular process, we hook the tap points of `task_struct` creation at `0xc102c8be` and deletion at `0xc102c7cc` by replacing them with an "`int 3`" instruction to trap to the hypervisor layer. Then we hook the tap point "`0xc125e3b1:mov 0x254(%edi),%esp`" to get the `task_struct` of the to-be-executed process from `edi` and then we count its CPU execution time from this moment to the next context-switch point. We keep a hash table to store the accumulated CPU time for each process, and meanwhile we store their `PID` field. Then right after user running `ps` to get the inside view, we also print the list of the live process with the `PID` and their CPU `TIME`. If there is a discrepancy, it indicates there is a hidden process. We can notice while attacker can change/forge all the PID field, it is impossible for them to forge the correct CPU `TIME` to mislead the outside view. That is why we call `TIME` is a tamper-proof attribute for a particular process.

Experimental Result. We have implemented the above detection algorithm in KVM-2.6.37 and tested with a guest Linux kernel 2.6.32.8. We only need to hook 3 tap points: creation, deletion and field propagation read. We used three rootkits to test our detection capability. As show in Table 5, these rootkits cover all the three basic tricks to hide a particular process. Through our cross view comparison, we have successfully detected all of these hidden processes.

Regarding the performance impact of our rootkit detector, we used a set of benchmarks including SPEC2006, Apache, and 7zip to evaluate the performance overhead introduced by our detection at KVM hypervisor layer, and we compared the results on the Native-KVM and our Tapping-KVM. As expected, there is not noticeable performance overhead for these benchmarks due to our lightweight instrumentation at the hypervisor layer. We measured that the average overhead for them is about 2.7 %.

6 Limitations and Future Work

First and foremost, AutoTap uses dynamic analysis to uncover the tap points and its effectiveness relies on the coverage of the dynamic analysis. Therefore, any kernel path coverage techniques (e.g., guided-fuzzing) would improve AutoTap. On the other hand, we can also notice that sometimes an incomplete coverage can still lead to a complete uncovering of the tap points. For instance, as shown for the `task_struct` creation/deletion tap points, while we may not be able to exercise all the kernel path and find out all (context-sensitive) tap points, we can notice that these tap points all eventually become context-insensitive and we can just use the `PC` that creates and deletes the `task_struct` instance as the tap point.

Second, currently AutoTap only reveals the object creation and deletion, field read, and object traversal tap points and demonstrates their use cases. We believe in addition to these tap points, there will be also other useful ones. Another future effort is to uncover more tap points and investigate new applications. A possible immediate future work is to identify the hot (or cold) read/write field tap points, namely, frequently read/write field, which might be useful to identify the likely-invariants (e.g., a field never gets changed) of object field. The other possible use case is to detect the hidden socket by using our tap points.

Third, when kernel has address space layout randomization (ASLR) enabled (note that since kernel version 3.14, Linux began to randomize kernel address space), the tap points we discovered from dynamic execution might not work in other executions. An immediate fix for this problem is to integrate our recent kernel ASLR derandomization effort [16], which exploited using various signatures from kernel code and data to derandomize the kernel address space.

Finally, while we have demonstrated our techniques working for Linux kernel, we would like to validate the generality of our system with other kernels. We plan to extend our analysis to FreeBSD, since it is also open source and we can validate our results easily. Eventually, we also would like to test our system with the closed source OS kernel such as Microsoft Windows. These are other future works.

7 Related Work

Tap Points Uncovering. Recently, Dolan-Gavitt *et al.* [10] presented TZB, the first system that can mine (`memgrep`) the memory access points for user level applications, to identify the places for active monitoring. While TZB and AutoTap share similar goal (TZB directly inspires AutoTap), we focus on different applications and use different techniques. Specifically, TZB focused on the user level applications such as web browser, whereas AutoTap exclusively focused on OS kernel. TZB starts from visible strings (`memgrep` type of approach can apply here), whereas AutoTap faces diversified, many non-string data structures in OS kernel and it starts from syntactic type of kernel object and then semantic type and then execution context to eventually derive the tap points for introspection.

Data Structure Reverse Engineering. Over the past decade, there are significant efforts on data structure reverse engineering, or more broadly type inference with executables [7]. Earlier attempts include aggregate structure identification (ASI) [23], value set analysis (VSA) [3,24]. Recently, Laika [9], REWARDS [22], TIE [20], Howard [26], Argos [30], and PointerScope [31] all aim to infer the (certain) data structure types from binary code. To infer the semantic type of data structures, while AutoTap uses the basic approach proposed in REWARDS, it extends it to OS kernels. Also, it combines other knowledge such as the data structure definitions for kernel driver development to resolve more semantic types, because of the large amount of point-to related kernel data structures. However, REWARDS only uses the type of arguments and return values from standard libraries for the inference.

Virtual Machine Introspection. VMI [15] is a security analysis technique that pushes the traditional in-box analysis into the outside hypervisor layer. It has been proposed as an effective means for kernel rootkit detection (e.g., [8,11,12,17] and malware analysis (e.g.,. [19,25]). While there are a number of efforts of using VMI or memory analysis technique (e.g., [5,8,21]) for hidden process detection (e.g., [11, 17,18]), in this work we enrich these knowledge with a tamper-proof approach by applying the tap points related to process descriptor and build a robust hidden process detection tool.

8 Conclusion

We have presented AutoTap, the first system that can automatically uncover the tap points of kernel objects of introspection interest from kernel executions. Specifically, starting from the interface of system call, the exported kernel APIs, and the data structure definitions for kernel driver developers, AutoTap automatically tracks kernel objects, resolves their kernel execution context, and associates the accessed context with the objects, from which to derive the tap points based on how an object is accessed. The experimental results with a number of Linux kernel binaries show that AutoTap is able to automatically uncover all the possible observed tap points for a particular type of object, which would be very challenging to achieve with manual analysis. We have applied the tap points uncovered by AutoTap to build a novel hidden process detection tool that can capture all the existing attacks including the DKOM based with only 2.7 % overhead on our tested benchmarks.

Acknowledgement. We thank the anonymous reviewers for their invaluable feedback. This research was partially supported by AFOSR under grant FA9550-14-1-0119 and FA9550-14-1-0173, and NSF CAREER award 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., Reps, T. Analyzing memory accesses in ×86 executables. In: CC, March 2004
4. Bauman, E., Ayoade, G., Lin, Z.: A survey on hypervisor based monitoring: approaches, applications, and evolutions. ACM Comput. Surv. **48**(1), 10:1–10:33 (2015)
5. Bianchi, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Blacksheep: detecting compromised hosts in homogeneous crowds. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012), Raleigh, North Carolina, USA, pp. 341–352 (2012)
6. Bovet, D., Cesati, M.: Understanding The Linux Kernel. Oreilly & Associates Inc., Sebastopol (2005)
7. Caballero, J., Lin, Z.: Type inference on executables. ACM Comput. Surv. **48**(4), 65:1–65:35 (2016)
8. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: The 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, IL, USA, pp. 555–565 (2009)
9. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI 2008), San Diego, CA, pp. 231–244, December 2008
10. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: mining memory accesses for introspection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2013)
11. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 297–312 (2011)
12. Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.: Robust signatures for kernel data structures. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, Illinois, USA, pp. 566–577. ACM (2009)
13. Fu, Y., Lin, Z.: Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: Proceedings of 33rd IEEE Symposium on Security and Privacy, May 2012
14. Fu, Y., Lin, Z., Brumley, D.: Automatically deriving pointer reference expressions from executions for memory dump analysis. In: Proceedings of the 2015 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2015), Bergamo, Italy, September 2015
15. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings Network and Distributed Systems Security Symposium (NDSS 2003), February 2003
16. Gu, Y., Lin, Z.: Derandomizing kernel address space layout for introspection and forensics. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. ACM, New Orelans (2016)
17. Jiang, X., Wang, X., 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 2007), Alexandria, Virginia, USA, pp. 128–138. ACM (2007)

18. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using lycosid. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008), Seattle, WA, USA, pp. 91–100. ACM (2008)
19. Lanzi, A., Sharif, M.I., Lee, W.: K-tracer: a system for extracting kernel malware behavior. In: Proceedings of the 2009 Network and Distributed System Security Symposium, San Diego, California, USA (2009)
20. Lee, J., Avgerinos, T., Brumley, D., TIE: principled reverse engineering of types in binary programs. In: NDSS, February 2011
21. Lin, Z., Rhee, J., Zhang, X., Xu, D., Jiang, X. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS 2011), San Diego, CA, February 2011
22. Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010), San Diego, CA, February 2010
23. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: POPL, January 1999
24. Reps, T., Balakrishnan, G.: Improved memory-access analysis for $\times 86$ executables. In: CC, March 2008
25. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: Proceedings of the 4th ACM European conference on Computer systems (EuroSys 2009), Nuremberg, Germany, pp. 47–60 (2009)
26. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS 2011), San Diego, CA, February 2011
27. Sumner, W.N., Zheng, Y., Weeratunge, D., Zhang, X.: Precise calling context encoding. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, (ICSE 2010), Cape Town, South Africa, vol. 1, pp. 525–534. ACM (2010)
28. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, Illinois, USA, pp. 545–554 (2009)
29. Zeng, J., Fu, Y., Lin, Z. Pemu: a pin highly compatible out-of-VM dynamic binary instrumentation framework. In: The 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE 2015), Istanbul, Turkey, March 2015
30. Zeng, J., Lin, Z.: Towards automatic inference of kernel object semantics from binarycode. In: Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2015), Kyoto, Japan, November 2015
31. Zhang, M., Prakash, A., Li, X., Liang, Z., Yin, H.: Identifying and analysing pointer misuses for sophisticated memory-corruption exploit diagnosis. In: NDSS, February 2012