Automatically Deriving Pointer Reference Expressions from Binary Code for Memory Dump Analysis

Yangchun Fu[†], Zhiqiang Lin[†], and David Brumley[*]

[†]The University of Texas at Dallas, Dallas, USA

[*] Carnegie Mellon University, Pittsburgh, USA

ABSTRACT

Given a crash dump or a kernel memory snapshot, it is often desirable to have a capability that can traverse its pointers to locate the root cause of the crash, or check their integrity to detect the control flow hijacks. To achieve this, one key challenge lies in how to locate where the pointers are. While locating a pointer usually requires the data structure knowledge of the corresponding program, an important advance made by this work is that we show a technique of extracting address-independent *data reference expressions* for pointers through dynamic binary analysis. This novel *pointer reference expression* encodes how a pointer is accessed through the combination of a base address (usually a global variable) with certain offset and further pointer dereferences. We have applied our techniques to OS kernels, and our experimental results with a number of real world kernel malware show that we can correctly identify the hijacked kernel function pointers by locating them using the extracted pointer reference expressions when only given a memory snapshot.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

Kernel Integrity, taint analysis, memory forensics

1. INTRODUCTION

A pointer, whose value is a memory address, is ubiquitous in a large body of software especially those written in C/C++. Recognizing and locating pointers in a memory (crash) dump is valuable in many applications. In program debugging, pointers are the root cause of segmentation fault [14, 15]. Given a crash dump, if we can locate where the crashed pointer is, it will significantly help the bug reporting. In security, pointers especially the ones pointing to program code (i.e., function pointers), are often the direct targets for control flow hijacks [20]. For instance, over 96% of kernel rootkits hijack kernel function pointers to subvert normal control flow of the OS kernel [24]. Given a running program or an OS kernel, if we can locate its function pointers, we would have been able to check their integrity and detect the control flow violations.

However, current practice to locate a pointer in memory requires the data structure knowledge of the corresponding program. Unfortunately, there are scenarios where data structures are not always available. For instance, for closed source software such as Microsoft Windows kernel, it is very unusual to have the complete kernel data structure definitions. Second, even with data structure definitions, if there is any ambiguous type (e.g., `void` pointer, or `union` type), it will stop traversing the target type (unless with further analysis to resolve the target type as in KOP [8] and MAS [11]), and may not be able to locate all pointers (e.g., the pointers stored in the `void` target type). Therefore, if we can directly locate pointers without any data structure definitions, such a technique would be of great practicality.

To advance the state-of-the-art, in this paper we introduce a new concept called *pointer reference expression* (ptr-rexp for short) and we show such an expression can be extracted from binary code and used to locate pointers in memory. More specifically, ptr-rexp encodes how a pointer is accessed through the combination of a base address (usually a global variable) with certain offset and further pointer dereferences. With ptr-rexp, we can then traverse a memory dump by following from the root of the pointer (e.g,. a global variable that is static) to reach the target locations. To derive ptr-rexp, we present a new dynamic binary analysis that tracks the dependences of how a memory address is computed. This analysis starts from a pointer data-use point (e.g., an indirect function call), and backward resolves the dependences until reaching the root of the pointer, namely a global variable address. Such a resolution process can directly produce a run-time address-independent ptr-rexp since global address is usually static, which can be used for cross checking.

As an application of our techniques, we demonstrate how to use it for kernel memory dump analysis, especially for the checking of kernel function pointer integrity. To this end, we have to solve another challenge: how to determine whether a pointer is hijacked. We propose a *pointer integrity checking* technique. We base our technique on the observation that after a program is compiled, the instructions (i.e., the *code*) are usually static, and the difference between the same program on two machines is the program *data*. As such, with our address-independent ptr-rexp, we can simultaneously traverse two memory snapshots: one is from a trusted kernel and the other is from the untrusted one. While the identified function pointers could be located in the dynamically allocated program addresses, which

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08
http://dx.doi.org/10.1145/2786805.2786810

can differ simply due to the behavior of the program heap allocators, fortunately our ptr-rexp is exactly designed to enable appropriate pointer integrity comparison between an untrusted kernel and the trusted one, and we can compare either their values or their targeted code to determine whether it has been hijacked.

We have implemented our proposed techniques in a prototype called FPCK. In addition to kernel dump analysis, FPCK will be useful in many other applications especially in security. For instance, it can be used to regularly inspect the integrity of kernel function pointers. In addition, it can also be used by Infrastructure as a Service (IaaS) cloud providers to manage the guest OSes with *kernel integrity check as a service*. Meanwhile, it can also be used in memory forensics to investigate the intrusions and estimate the damages caused by kernel malware.

In short, the main contributions of this paper are:

- We present a set of novel techniques to automatically locate function pointers in a memory snapshot. The pointer location is encoded by a static *pointer reference expression* that is derived by a dynamic data dependence analysis.
- We apply our techniques and develop a binary exclusive out-of-VM approach to automatically check the integrity of kernel function pointers hijacked by kernel malware.
- We have implemented these techniques in our prototype FPCK. Our experimental results with Linux kernel show that with respect to the tested malware, FPCK successfully identifies all of the hijacked kernel pointers within just a few seconds.

2. BACKGROUND AND OVERVIEW

Objective. Intuitively, when given a memory dump, it often requires the corresponding data structure definitions to traverse the memory and reach the data of interest. However, data structure knowledge is not always available. The goal of this paper is to demonstrate the practicality of traversing a memory dump to reach data of interest without accessing the program source code or data structure definitions. More specifically, we show there exists techniques that can automatically derive ptr-rexp from program executions, and such expressions can be used for cross checking the integrity of kernel function pointers.

The reason to focus on kernel function pointer is because kernel rootkit, a piece of malware designed to tamper with kernel behavior, often hijacks kernel function pointers to subvert normal control flow of the OS kernel [24]. Hijacking kernel function pointer allows kernel malware to conceal the presence of malicious behavior, open back-doors for future intrusion, steal private data such as the keystrokes, elevate privileges of other malicious processes, and disable defenses (as those discussed in [6,8,12,13,21,22,26,28,31]). Therefore, it is often used by many cyber attacks due to its root level privilege and high stealthiness, including Stuxnet, Duqu, and the recent RSA SecurID compromise.

Several prior works such as KOP [8] and OSck [18] have demonstrated that we can periodically take the snapshot of the guest OS running atop virtual machines (VM) and then traverse its memory snapshot to perform the integrity check. However, KOP and OSck require the access of kernel source code to build a data structure graph, from which to traverse the kernel function pointers. Unfortunately, in many cases end-users including cloud providers only have the binary code of an OS kernel. Thus, it is imperative to have a binary exclusive technique that can pinpoint kernel function pointers in memory snapshot such that we can check their integrity correspondingly.

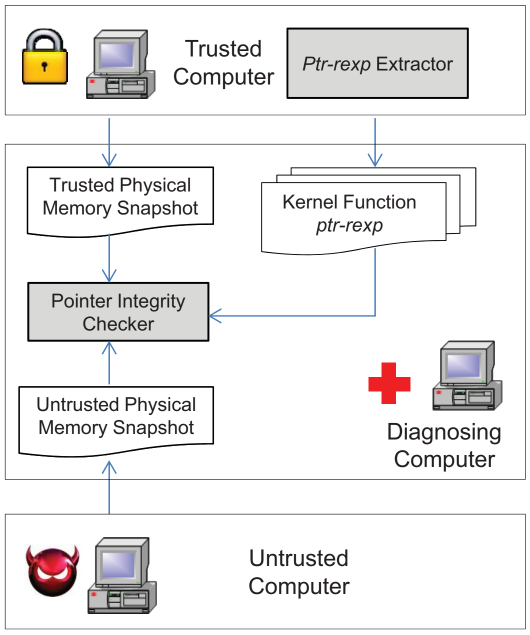

Figure 1: An overview of FPCK. All the involved computers could be the real physical machines, virtual machines, or their combinations.

Challenges. Locating the hijacked kernel function pointers given only a physical memory snapshot is challenging for two reasons:

- **Where to locate the kernel function pointers**. Kernel function pointers can be located in many places, such as in kernel global data regions and kernel heap regions. It might be possible to identify the hijacks of kernel global function pointers by aligning the virtual address of the memory snapshot of a untrusted machine (denoted M_u) and a trusted machine (denoted M_t). However, it is challenging to locate the function pointers in other memory regions such as those in the kernel heap.
- **How to determine whether they are hijacked**. Assume we are able to locate the memory addresses of the corresponding kernel function pointers, we cannot merely compare their values in the memory cell between M_u and M_t to check the hijacks, as the value of a function pointer could be different since a kernel function can be loaded into different memory addresses in different running instances of the same kernel, which is especially true for kernel modules. Of course, for the core kernel code that is not loaded as kernel modules, the values of their function pointers should be identical (e.g., the values in system call table), and we can compare their values to identify the hijacks.

Overview. To address the above challenges, we have designed an array of *binary exclusive* techniques in our prototype FPCK. As illustrated in Fig. 1, there are two key components inside FPCK: a ptr-rexp *extractor*, and a pointer value *comparer*. Given a trusted machine T, our *extractor* analyzes its execution and derives a static data reference expression, namely ptr-rexp, for each observed kernel function pointer. This expression reflects how it should be accessed starting from the kernel global locations. Next, whenever we want to check the integrity of the kernel function pointers of an untrusted computer, we first take a snapshot of its physical memory M_u. Then we run the pointer integrity *checker* on our diagnosing machine, which will scan whether U's kernel function pointers have been hijacked. The *checker* detects hijacks by performing a low-level pointer value or pointer target comparison between U and T. Each mismatch between M_u and M_t indicates a potential hijack. In the next two sections, we present the detailed design of the two key components of FPCK.

Scope and Assumptions. We assume we are able to take the memory snapshot of the running VMs which are running atop x86 architectures. Along with the snapshot, we also assume we can acquire the value of the page table base register (i.e., CR3 in x86), through which to perform the virtual to physical address translation. In addition, we assume the kernel code pages are not writable and the code itself is identical across different VMs for the same version. This is reasonable given that (public/private) cloud providers often offer users with pre-installed VMs. It is highly likely that these VMs tend to have the same kernel version. In addition, to check M_u, we assume that we have the corresponding trusted kernel copies (including the corresponding kernel modules) of the running guest OSes. Note that the specific kernel version of the guest OS can be determined by other approaches (e.g., [16, 17, 25]). Regarding the workloads running on the trusted or untrusted machines, there is no constraint and they can be different.

Also, similar to all other snapshot-based approaches, (e.g,. [4, 5, 8, 11, 18]), we are only interested in the persistent function pointers (continuing to exist over the entire life span), and will exclude these temporary pointers. The reason is that snapshot based approach is context-less (a memory snapshot can be taken at arbitrary time), and we cannot reliably check the integrity of these temporary ones. While this is a limitation in general for all these snapshot-based approaches, fortunately over 96% kernel rootkits tamper with persistent function pointers according to the survey by Petroni et al. [24]. Meanwhile, many kernel function pointers are actually persistent, e.g., 97% of function pointers in Microsoft Windows does not change over their entire life span according to the result from HookScout [35].

3. PTR-REXP EXTRACTOR

The goal of ptr-rexp *extractor* is to (i) derive the memory addresses into static expressions for the exercised kernel function pointers (§3.1), and (ii) prune non-persistent function pointers and merge multiple values for the same pointer if there is any (§3.2). In the following, we present the detailed design of each of these techniques.

3.1 Deriving ptr-rexp

Overview. Being one of the key techniques of FPCK, ptr-rexp derivation aims to to produce an address-independent expression of each function pointer used in an executed control path. Then during the integrity checking phase, this expression allows us to traverse and compare the function pointers between trusted kernel memory M_t and untrusted kernel memory M_u. If they differ, then we say that kernel malware has changed the value of the function pointer. For example, the following code shows how the `read` operation is calculated from a data structure in Linux kernel 2.6.08:

```
1. mov 0xc034bc78(), %eax; move proc_root_fs to eax
2. mov 0x20(%eax), %eax  ; proc_root_fs->f_ops
3. <... irrelevant instrs ...>
4. call 0x8(%eax)     ; proc_root_fs->f_ops->read
```

The example shows that the final call address at line 4 is derived from the memory value stored at 0xc034bc78 (which is a kernel global variable). A typical approach used by kernel malware is to tamper with the value at 0xc034bc78, or *(0xc034bc78)+ 0x20, or *(*(0xc034bc78)+ 0x20)+ 0x8 (note that $*$ represents the pointer dereference) such that the malicious code is called at line 4. Unfortunately, the value stored at these memory addresses may change between different instances of the kernel. Thus, we cannot simply compare the absolute memory address calculated at line 4.

Our main idea is to use a ptr-rexp of an indirect jump or call to uniquely represent the intended target. Our ptr-rexp encompasses all calculations used to determine an indirect jump or call target. In the above example, the ptr-rexp is: $*(*(*(0xc034bc78)+0x20)+0x8)$. The intuition why the ptr-rexp works is that while the *data stored* at addresses may differ between M_u and M_t, the *code* to calculate the location will be the same (recall that we assume kernel code cannot be tampered by malware, and hypervisor can often enforce this).

Detailed Algorithm. At a high level, given an indirect call or jump to an address α, we present an algorithm to recursively walk the list of instructions used to calculate α to build up our ptr-rexp. The main intuition behind our algorithm is that the roots of data dependence are pre-determined addresses. While the value of a memory cell used to calculate a jump/call target may change, the location does not. We recognize the potential locations by looking for literal values that fall within the kernel address space. We "taint" [23] such values, and perform dynamic data dependence tracking on any instruction derived from a tainted value. This is different compared to other techniques such as HookScout [35] where it taints the known values of kernel function pointers and tracks how these values propagates, whereas we track how a pointer is computed as described below. We also keep a data structure that maintains the semantics of each instruction that operates on tainted data (which we call tainted instructions). The algorithm walks the data structure to generate the final expression on-the-fly when an indirect call or jump target is tainted.

type operand = Reg **of** name | Mem **of** addr
type shadow = (operand, PC) **Hashtbl**
type instMap = (PC, instRecord) **Hashtbl**
type instRecord = (I-semantics, taintOp, taintOp)
type I-semantics = Move | Binary | Call-Mem | …
type regTaint = (V, PC_p)
type taintOp =
  MemOpTaint **of** regTaint × regTaint × scale × disp
  | RegOpTaint **of** regTaint
  | NoOpTaint

Figure 2: The data structure types used in FPCK.

The type signatures used by our algorithm are presented in Figure 2. The first data structure is of type operand that can be either a register or a memory address. Note that the literal value operand will be recognized and taken special care. The second data structure is of type shadow that maps an operand to a program counter (PC). We use `S` to denote an instance of shadow. For brevity, we write `S[0xff]` and `S[eax]` to mean `S[Mem[0xff]]` and `S[Reg[eax]]`, respectively. `S` is used to maintain two invariants in our algorithm:

- ***Tainted Invariant:*** For efficiency, we only track instructions dependent upon literal values (or the so called immediate values) that fall within kernel address space. Such literal values can be bases in the data dependence calculation for kernel function pointers. Specifically, given an instruction $PC : l := e$ where PC is the program counter, l is the assigned memory

Table 1: Example code for the shadow record creation and propagation.

PC: Machine Code	Assembly Code	Partial Program States (Registers, Memory)	Shadow S[Operand]=PC
0xc0106bc0: 68 10 4d 11 c0	push 0xc0114d10;	ESP=0xcf581004, M(0xcf581004)=0xc0114d10	S[0xcf581004]=0xc0106bc0
...	;//push do_page_fault		
0xc0106a02: 8b 7c 24 20	mov 0x20(%esp),%edi	ESP=0xcf580fe4, EDI=0xc0114d10	S[EDI]=S[0xcf581004] =0xc0106bc0
...			
0xc0106a1b: ff d7	call %edi	ESP=0xcf580fe0, EDI=0xc0114d10	S[EDI]=0xc0106bc0
...	;//call do_page_fault		
0x0c015f94: ff 14 85 3c 86 28 c0	call 0xc028863c(, %eax,4)	EAX=4, M[0xc028864c]=0xc0144d8a	S[0xc028864c]=0x0c015f94
	;//call sys_write		
0xd894e007: a1 78 bc 34 c0	mov 0xc034bc78(), %eax	EAX=0xd7fee2e0	S[EAX]=0xd894e007
	;// proc_root_fs		
0xd894e00c: 8b 40 20	mov 0x20(%eax), %eax	EAX=0xc028ea80	S[EAX]=0xd894e00c
	;// proc_root_fs->f_ops		
0xd894e013: ff 50 08	call 0x8(%eax)	EAX=0xc028ea80,M[0xc028ea88]=0xc015ed7e	S[0xc028ea88]=0xd894e013
	;// proc_root_fs->f_ops->read		

cell or register, and e is derived from a literal falling in kernel space, then `S[`l`]` is defined. This is a type of taint tracking where `S[`l`]` is defined iff e is tainted.

- ***PC Invariant:*** During the ptr-rexp extraction we walk a log of tainted instructions to compute the jump/call target expression. Given PC: $l := e$, we maintain the overall invariant with two minor invariants on tainted values e. First, if e is a memory load `M[v]` for value v, then `S[`l`] = S[v]`. This invariant ensures that each tainted memory load maps to the definition site where the memory cell was initialized with a tainted value. Second, when e is not a memory load `S[v]` points to the definition site for each e, including such as the definition site for a tainted register. Overall, these two invariants ensure that (i) `S` points to the definition site for a tainted memory cell, and (ii) `S` points to the definition site for tainted registers. Note that while the above invariants can be generalized to any expression, currently we implement the data structures for x86. In particular, each e can have at most one memory reference. The memory reference is based upon at most two registers of the form:$r_1 + r_2 * scale + disp$.

The above invariants allow us to recursively determine the PC for all instructions used to calculate a tainted indirect jump or call target. We associate PC values to specific instruction semantics via the instMap record, as illustrated in Figure 2. Each PC value defined in `S` is also mapped to a record in an instMap. The records of type instRecord consist of a semantic type I-semantics for the operation (Move, Binary operation, Call, etc.), and two taint records for the two operands of type taintOp. For simplicity, we only describe at most two source expressions (regOpTaint or memOpTaint) for an executing instruction. Note that an instruction could have only one operand (instead of two) or no operand (that is why we introduce NoOpTaint). In addition, for the regTaint of regOpTaint, we use a 2-tuple (V, PC_p), which denotes the taint dependency of the register operand for an instruction where V is the concrete value if there is no dependency (otherwise V is 0), and PC_p tracks the *previous* PC that generates the propagated taint record. We provide detailed examples to illustrate why we define our data structure in this way in the following based on how a general taint analysis works.

Taint Sources Our shadow record (including type shadow and instRecord, as well as the mapping instMap) is generated using two rules: either (1) an instruction which generates a data definition (such as memory or register write), or (2) an instruction which has a memory operand that involves a global memory address or its propagation.

Rule-I. Similar to all other taint analyses, a data definition for an operand (Reg or Mem) is generated by (i) a data movement instruction, such as MOV/PUSH/POP, etc., or (ii) a data arithmetic instruction, such as ADD/SHL/AND, etc. Based on the instruction semantics, we accordingly generate our taint record for the corresponding operand.

Example. As shown in the first row of Table 1 for instruction `0xc0106bc0: push 0xc0114d10`, it pushes a literal value `0xc0114d10` which falling within the kernel address space (we hence classify this value is a reference to a global variable) to memory address 0xcf581004 (pointed by `esp`), then we generate a shadow record for memory address `0xcf581004` with PC `0xc0106bc0` (i.e., S[0xcf581004] = 0xc0106bc0), and an instRecord for the source operand as ((0,0), (0,0), 0, 0xc0114d10) which is indexed by PC 0xc0106bc0 as shown in the first row of Table 2. The destination operand (pointed by register `esp`) under current taint context does not have a global address dependency, whose memOpTaint is hence ((0,0), (0,0), 0, 0) or NoOpTaint.

Rule-II. This is a special rule in our data dependence tracking. Specifically, we will also generate an instRecord whenever an instruction has a Mem operand, whose address is directly or transitively derived from a kernel global memory address. In particular, in the x86 architecture, a memory address for a Mem operand is computed using the following formula,

$$Displacement(BaseAddr, Index, Scale) = \\ BaseAddr + Index \times Scale + Displacement$$

where $BaseAddr$ is the Reg that has the starting address or base address of the accessed memory, $Index$ is the Reg used to determine the offset from the base address, $Scale$ is the data size based multiplier for the $Index$, and $Displacement$ (often an immediate value) is the additional offset adjustment from the $BaseAddr$. If the Reg of the $BaseAddr$ is tainted (i.e., the address is transitively derived from a global variable), or the $Displacement$ is a global address, we will generate a new instRecord, which captures the dependences on how an address is computed.

Example. Consider `call 0xc028863c(,%eax,4)` in the second code snippet in Table 1. This instruction is interesting because it calls a system call routine `sys_write`. The Mem operand for this instruction is computed from an empty $BaseAddr$ Reg (thus no taint), an $Index$ Reg `eax` with a value 4 (thus no global address dependency), a $Scale$ with 4, and the displacement value `0xc028863c` that is a kernel global memory address. Therefore, from the instruction semantics (Call-Mem), we can infer that the memory address generated from this instruction is a global mem-

Table 2: Exampes of our instRecord for executed instructions in Table 1.

PC	*I*-semantics	1st Operand memOpTaint (regTaint, regTaint, Scale, Disp)				2nd Operand memOpTaint (regTaint, regTaint, Scale, Disp)			
		(V, PC_p)	(V, PC_p)	Scale	Disp	(V, PC_p)	(V, PC_p)	Scale	Disp
0xc0106bc0	PUSH-IMM	(0,0)	(0,0)	0	0xc0114d10	(0,0)	(0,0)	0	0
0x0c015f94	CALL-MEM	(0,0)	(0x4,0)	0x4	0xc028863c				
0xd894e007	MOV-M2R	(0,0)	(0,0)	0	0xc034bc78	(0,0)	(0,0)	0	0
0xd894e00c	MOV-M2R	(0,0xd894e007)	(0,0)	0	0x20	(0,0)	(0,0)	0	0
0xd894e013	CALL-MEM	(0,0xd894e00c)	(0,0)	0	0x8				

ory address reference which refers to `0xc028863c + 4×4 = 0xc028864c`, and we will consequently generate a shadow record as

$$S[0xc028864c] = PC_{call} = 0x0c015f94$$

and an instRecord with instMap as

0x0c015f94 → (CALL-MEM, ((0,0), (0x4,0), 0x4, 0xc028863c), NoOpTaint)

as presented in the second row of Table 1 and Table 2, respectively.

Taint Propagation The shadow record will be flowed to the destination shadow of the corresponding Reg or Mem operand. The instRecord will not, but it will be mapped to the shadow by instMap. Such a design saves the shadow memory space and improves the performance of our data dependence tracking (because of the indirection we introduced), compared to the normal method that also propagates the instRecord. Also, when propagating the shadow record, if the source operand generates a new instRecord (**Rule-II**), we will not propagate the original shadow record, but rather propagate the newly generated one.

Example. For instance, as shown in the third example code snippet in Table 1, for `0xd894e00c: mov 0x20(%eax), %eax`, we will not directly propagate the shadow of the *BaseAddr* Reg `eax` which is

S[EAX] = 0xd894e007 → (MOV-M2R, ((0,0), (0,0), 0, 0xc034bc78), ((0,0), (0,0), 0,0)))

and instead we will first create a new instRecord with instMap as

0xd894e00c → (MOV-M2R, ((0, 0xd894e007), (0,0), 0, 0x20), ((0,0), (0,0), 0,0)))

and propagate its shadow to `eax` (S[EAX] = 0xd894e00c).

Taint Sinks We build the ptr-rexp on-demand from the data dependency and the instruction semantics maintained in our data structures, namely shadow, instRecord, and instMap. At a high level, we generate the ptr-rexp by walking back through our instRecord Hashtbl guided by instMap, the walk recursively resolves each expression *e* used in the right-hand side of an assignment to the original tainted data definition. The detailed algorithm is presented in Algorithm 1.

In particular, the ptr-rexp generation algorithm takes in an instruction address *p*, a value *v* for a tainted operand, and a global instMap *t*. As a base case (line 2), when the address is 0 (PC_p=NULL), our resolution is complete and we return *v*. Note that the base case is reached when taint is first introduced. To see this, we only add 0 to the instRecord at such sites. The algorithm recursively walks the instRecord (indexed by PC) and produces a final expression for the tainted operands. In the following, we present a representative example to show how we perform the address resolution.

Algorithm 1: On-demand ptr-rexp Generation

```
1:  let rec resolve_data_path (p: PC) (v: value) (t: instMap): exp =
2:    if p = 0 then (Value(v)) else (
3:      let (sem, op1, op2) = Hashtbl.find t p in
4:        match sem with
5:          Move -> resolve_op p op1 t
6:          | Binary -> BinOP(resolve_op p op1 t, resolve_op p op2 t)
7:          | Call-Mem -> resolve_op p op1 t
8:    )
9:  and resolve_op (p: PC) (op: taintOP) (t: instMap): exp =:
10:   match op with
11:     memOpTaint ((v1, pc1), (v2, pc2), scale, disp) ->
12:         let regValue1 = resolve_data_path pc1 v1 t in
13:         let regValue2 = resolve_data_path pc2 v2 t in
14:         DeRef (regValue1, regValue2, scale, disp)
15:   | regOpTaint (v3, pc3) -> ( resolve_data_path pc3 v3 t )
16:   | NoOpTaint -> Value (0)
```

Example. Consider the third example in Table 1 again. When the instruction `0xd894e013: call 0x8(%eax)` is executed, the targeted memory address is 0xc028ea88 (as EAX = 0xc028ea80). Because this instruction generates a new memory reference, we will have a new instRecord for address 0xc028ea88 as

S[0xc028ea88] = 0xd894e013 → (CALL-MEM, ((0,0xd894e00c), (0,0), 0, 0x8),()).

Since the PC_p in the *BaseAddr* Reg is 0xd894e00c and not NULL (as shown in the 5th row in Table 2), we recursively resolve the operand (resolve_op) with PC=0xd894e00c and a memOpTaint. Note that according to the instMap, the instRecord indexed by 0xd894e00c is

(MOV-M2R, ((0,0xd894e007), (0,0), 0, 0x20), ((0,0), (0,0), 0, 0)).

Now PC_p is 0xd894e007 (also not NULL, and we have to recursively resolve the operand with PC=0xd894e007 and a memOpTaint), and the instRecord indexed by 0xd894e007 is

(MOV-M2R, ((0,0), (0,0), 0, 0xc034bc78), ((0,0), (0,0), 0, 0)).

Now that the PC_p is NULL, we will directly output a Value v which is 0 and return. Because of the recursion, a DeRef will be called and it will dump a ptr-rexp of the function pointer as *(*(*(0xc034bc78)+ 0x20)+ 0x8). Using this ptr-rexp, we can precisely locate the function pointer at the untrusted memory M_u. Note that this example is actually the `read` function for the `proc` file system in Linux kernel-2.6.08. From this example, we see that although `read` may be dynamically loaded and the structure pointer `f_ops` could be in kernel heap, we can still locate it precisely in an untrusted kernel memory by using its ptr-rexp.

3.2 Handling Practical Issues

Handling Loops. Because of the use of recursive data structures, e.g., arrays, link lists, and trees, one PC can be used multiple times (with different targeted memory addresses) in a ptr-rexp. For instance, to remove a Linux kernel module, kernel will iterate each module descriptor to find the to-be-removed module, and then call `module.exit` function. During the iteration, instruction that iterates (`module.list.next`) will be used multiple times to reach the final desired module. Then, if we back track the dependence graph from the instRecord Hashtbl (examples shown in Table 2), we will not be able to find the unique path. As such, we introduce a counter to our PC to make the back-tracking unique and generate an abstract ptr-rexp for the recursive data structures. More specifically, all of our own data structure defined in Fig. 2 will not be changed except that the PC becomes $(PC, counter)$, where $counter$ is initially 0 and gets increased by 1 whenever encountering a new data dependence while iterating the loop.

However, directly using the above $(PC, counter)$ dependence graph would only capture one instance of ptr-rexp. Therefore, we would like to generate an abstract representation that captures all of its instances. To generate an abstract ptr-rexp, we need to build a path graph $G = (V, E)$, which captures the dependencies among the instructions that are used to calculate the address of the function pointer. Here node V denotes the instructions involved in the address calculation, and edge E denotes the dependencies between the instructions. G is built when we back track the instRecord. Then we traverse G to generate the abstract representation of the recursive ptr-rexp. Similar to regular expressions where a string can have one or multiple appearances, our abstract representation of the ptr-rexp can capture the cases where one or multiple dereferences are needed to reach a particular pointer instance. Details are elided due to space limitation.

Pruning the ptr-rexp of Non-persistent Pointers. It is possible that a function pointer can only exist in certain time window, e.g., kernel could allocate a dynamic object at certain context, and later remove it. We call this pointer non-persistent. For any snapshot based memory analysis, we have to exclude them because we cannot check them in a reliable way (unless with execution context information).

To prune these non-persistent pointers, our approach is quite straightforward. Specifically, we just take N number of memory snapshots running with diverse workloads to perform the cross check during the dynamic execution phase, which is also how HookScout [35] inspects whether a pointer is persistent. Currently, we set N as 100. If the pointer exists over all the memory snapshots (by traversing them through our ptr-rexp), then we keep it. Otherwise, we will not check these pointers.

Merging the Target Values for Mutable Pointers. Also, a persistent pointer can change its value during its life span. Therefore, when checking its integrity, we have to be aware that such a pointer could have a set of trusted values. To this end, we just merge these values whenever we observe there is a different trusted value in the trusted memory snapshot.

4. POINTER INTEGRITY CHECKER

Given a ptr-rexp of a kernel function pointer f, with a trusted kernel memory snapshot M_t and an untrusted kernel M_u, our pointer integrity *checker* aims to identify whether f has been compromised by comparing ptr-rexp(f, M_t) and ptr-rexp(f, M_u) where ptr-rexp(α, M) denotes the concrete address when following the ptr-rexp of α in snapshot M. There are two steps in this comparison.

Step-I: Direct Value Comparison for Core Kernel Code. If f points to core kernel code (which are those excluding kernel modules) and the value of ptr-rexp(f, M_u) belongs to the trusted values set from ptr-rexp(f, M_t), meaning that this pointer is not compromised in M_u, then we directly return since this pointer is trusted. Otherwise, if it points to kernel modules, goto **Step-II**. Otherwise, we return this pointer has been compromised.

```
OFFSET          TYPE            VALUE
0000000d        R_386_PC32      current_kernel_time
00000013        R_386_32        init_task
0000001c        R_386_32        init_task
00000031        R_386_32        .rodata.str1.1
...
```

(a) Relocation Table

```
 00000000 <init_module>:
  0:    55                          push   %ebp
  1:    57                          push   %edi
  2:    56                          push   %esi
  3:    53                          push   %ebx
  4:    83 ec 10                    sub    $0x10,%esp
  7:    8d 44 24 08                 lea    0x8(%esp),%eax
  b:    50                          push   %eax
  c:    e8 fc ff ff ff              call   d <init_module+0xd>
 11:    8b 1d 50 00 00 00           mov    0x50,%ebx
 17:    83 eb 50                    sub    $0x50,%ebx
 1a:    81 fb 00 00 00 00           cmp    $0x0,%ebx
 20:    8b 7c 24 08                 mov    0x8(%esp),%edi
 24:    8b 6c 24 0c                 mov    0xc(%esp),%ebp
 28:    74 21                       je     4b <init_module+0x4b>
 2a:    ff b3 88 00 00 00           push   0x88(%ebx)
 30:    68 00 00 00 00              push   $0x0
...
```

(b) Disassembly of the Static Code

```
0xd894e000: 55                      push   %ebp
0xd894e001: 57                      push   %edi
0xd894e002: 56                      push   %esi
0xd894e003: 53                      push   %ebx
0xd894e004: 83 ec 10                sub    $0x10, %esp
0xd894e007: 8d 44 24 08             lea    0x8(%esp), %eax
0xd894e00b: 50                      push   %eax
0xd894e00c: e8 43 e0 7c e7          call   0xc011c054
0xd894e011: 8b 1d 90 7a 28 c0       mov    %ds:-0x3fd78570(), %ebx
0xd894e017: 83 eb 50                sub    $0x50, %ebx
0xd894e01a: 81 fb 40 7a 28 c0       cmp    $0xc0287a40, %ebx
0xd894e020: 8b 7c 24 08             mov    %ss:0x8(%esp), %edi
0xd894e024: 8b 6c 24 0c             mov    %ss:0xc(%esp), %ebp
0xd894e028: 74 21                   je     0xd894e04b
0xd894e02a: ff b3 88 00 00 00       push   %ds:0x88(%ebx)
0xd894e030: 68 a4 e0 94 d8          push   $0xd894e0a4
...
```

(c) Disassembly of the Dynamically Loaded Code in Memory Snapshot

Figure 3: Code differences in static disk image and dynamic memory snapshot for the same function

To decide whether a pointer points to core kernel code or kernel module is trivial because we have the trusted kernel. In our analysis phase, we check each ptr-rexp and verify its target address. If it does not belong to kernel module, then we conclude its the core kernel code and directly compare the values.

Step-II: Direct Target Comparison for Kernel Modules. When f points to kernel modules, it becomes more sophisticated as we cannot compare its value with the trusted kernel any more. Instead, we have to compare its targeted code. Before presenting our solution, we would like to first examine how a kernel module is loaded, and what the code difference is in different executions. In general, kernel code is composed of: (1) static core kernel code, and (2) dynamically loaded kernel modules. Functions in dynamically loaded kernel modules may be loaded to different memory addresses. As a result, some of the operands of some instructions need to be dynamically patched when loading the modules. The patching is informed by the relocation table in the binary code generated by compilers. For example, as shown in Fig. 3(c) for a kernel module code snippet which is disassembled from a memory snapshot, compared to Fig. 3(b), we could find there are four operand-patches at relative offset 0xd, 0x13, 0x1c, 0x31, and these locations are specified in the relocation tables in the module's binary code. Such a dynamic patching mechanism works for both Windows and Linux kernel modules.

Table 3: The number of exercised reference path.

Kernel Version	Call-MEM	Call-REG	Jmp-MEM	Jmp-REG	Σ
2.6.08	1234	155	250	0	1639
2.6.13	1175	141	257	11	1584
2.6.24	1237	474	231	0	1942
2.6.28	1182	423	273	0	1878
2.6.30	1262	456	282	0	2000
2.6.32	1284	365	232	0	1881
2.6.33	1284	366	227	0	1877
2.6.34	1286	360	245	0	1891
2.6.35	1239	352	239	0	1830
2.6.38	1213	375	234	15	1837
3.0.0	1394	451	276	29	2150
Average	1254	398	250	5	1907

Therefore, our solution is to directly compare the code page of the targeted function body in both M_u and M_t. More specifically, since we have the starting address of the function (that is the pointer value) and the relocation table of the targeted module (that is from the trusted kernel), then we compare each un-patched byte (the patched byte is informed by the information stored in the relocation table) until the end of the function. We can know the end address of the target function, because we have the ground truth of the trusted kernel modules and they are not usually obfuscated (that also explains why FpCk rejects the unknown kernel modules). There could be some optimizations, such as only comparing the first x-bytes of the code. In our design, we just compare the entire function body.

5. EVALUATION

We have developed a prototype of FpCk with over 10K lines of our own C code. Among then, 8K lines of the code (LOC) belongs to ptr-rexp *extractor* which is built atop QEMU-1.01 [3] and the rest 2K LOC belongs to the pointer integrity *checker* that is an independent program, which scans the memory, normalizes the instruction byte in the target page, and performs the comparison. In this section, we present our evaluation result. We first tested the effectiveness of FpCk with a number of Linux kernels and a number of kernel rootkits in §5.1. We report the performance of each component of FpCk in §5.2. All of our experiments were carried out on a host machine with Intel Xeon W5620 CPU, 24G memory, running Red Hat Enterprise 6.2 with Linux kernel 2.6.32.

5.1 Effectiveness

Experiment Setup. We acquired an extensive test case suite from the Linux test project (LTP) [2] for our dynamic analysis. The LTP consists of a large set of regression tests designed to confirm the behavior of a Linux kernel. In total, we downloaded 2,173 test cases from LTP. Interestingly, the LTP test suite has several test cases for each specific aspect of Linux kernel, and we consider these cases as duplicates since they all test the same kernel functionality. Thus, we select only one test case from each functionality test suite to be part of our test cases. Also we limit each test case to be finished in 5 minutes. If not, we removed it from our final set of test cases. After applying these constraints, we end up with 440 test cases, which we package them in a script file and automatically execute them in the trusted OS.

Result. To test how effective our kernel ptr-rexp *extractor* is, we took 11 Linux kernels. As shown in Table 3, for each tested Linux kernel (installed with the default kernel modules or device drivers), we evaluated how many ptr-rexp we extracted from the indirect control flow transfers. Specifically, these indirect control flow transfers include indirect calls via a memory address (Call-MEM), indirect

Table 4: Kernel pointer check with Linux rootkits. "-" denotes there is no trusted value.

Rootkit	Symbol Name of the Pointer	Trusted Value	Hijacked Value	|C|
override	module->init	-	0xd0923ad6	2
	moduel->exit	-	0xd0923af7	2
	sys_read	0xc0144d27	0xd092343c	1
	sys_chdir	0xc0143ced	0xd0923001	1
	sys_getuid	0xc011f59c	0xd09232ce	1
	sys_geteuid	0xc011f5ac	0xd09232f1	1
	sys_getdents64	0xc0154292	0xd0923314	1
Synapsys-0.4	module->init	-	0xd09267e8	2
	module->exit	-	0xd0926896	2
	sys_fork	0xc010488a	0xd092651e	1
	sys_write	0xc0144d8a	0xd09265f6	1
	sys_open	0xc014444c	0xd0926000	1
	sys_kill	0xc0121fa5	0xd09264c5	1
	sys_clone	0xc01048a4	0xd092657f	1
	sys_getdents	0xc0154082	0xd09265e0	1
	sys_getuid	0xc011f59c	0xd09263f9	1
kbdv3	module->init	-	0xd091b1aa	2
	module->exit	-	0xd091b215	2
	sys_utime	0xc0143970	0xd091b000	1
	sys_getuid	0xc011f59c	0xd091b142	1
	sys_utimes	0xc0143b84	0xd091b097	1
phalanx-b6	sys_read	0xc0144d27	0xce271000	1
	sys_open	0xc014444c	0xcdde6000	1
	sys_newlstat	0xc014c7ad	0xcdde3000	1
	sys_lstat64	0xc014c9a8	0xcdde2000	1
	tcp4_seq_show	0xc022be91	0xcdde5000	1
adore-2.6	module->init	-	0xd8985000	2
	module->exit	-	0xd897f9b4	2
	ext3.ext3_readdir	dynamic	0xdd97f774	6
	do_sync_write	0xc0144bb0	0xd897f8a4	5
	proc_root_readdir	0xc016b608	0xd897f477	6
	proc_root_lookup	0xc016b5ba	0xd897f13e	6
rkit-1.01	module->init	-	0xd091b05d	2
	module->exit	-	0xd091b097	2
	sys_setuid	0xc0123209	0xd091b000	1
suckit-2	idt enty 0x80	0xc0105f68	0xcc8c0906	1
hookswrite	module->init	-	0xd08c3000	2
	module->exit	-	0xd0843216	2
	idt enty 0x80	0xc0105f68	0xd0843000	1
int3backdoor	module->init	-	0xd08a119c	2
	idt enty 0x3	0xc0106b48	0xd08a1000	1

call via a register (Call-REG), indirect jumps to a memory address (JMP-Mem), and indirect jumps via a register value (JMP-REG). We can see from Table 3 that our *extractor* has revealed thousands of reference expressions to reach those kernel function pointers, and the majority of them are from indirect Call-MEM cases. Since our *extractor* is dynamic analysis based, certainly it does not expose all of the kernel function pointers. However, it has exercised those commonly used, which are also usually of attackers' interest, as discussed below.

Checking the Function Pointer Integrity. We also evaluated *pointer value comparer* to check the function pointer integrity of Linux kernel 2.6.08. We took 9 real word rootkits from *packetstormsecurity.org* for this experiment. The detailed result is presented in Table 4. Note that we tweaked the code of Synapsys-0.4 and suckit-2 such that they can be compiled on our testing kernel. The rest were compiled and ran without any change.

For each rootkit, we present the symbol name of the function pointer in the 2^{nd} column of Table 4, the trusted value of the function pointer in the 3^{rd} column, and the hijacked value in the 4^{th} column. Finally, we present the length of the data reference chain (|C|) in the last column. As we can see from this table that for each rootkit FpCk successfully identified all the hijacked pointers without any false positives (FP) or false negatives (FN). Among these 41 hijacked pointers, 28 pointers are pointing to the core kernel code, and we

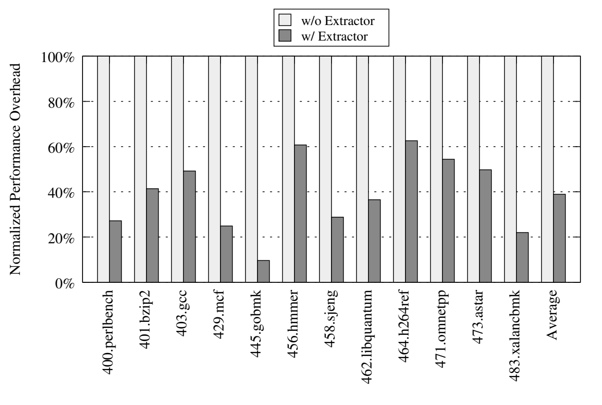

Figure 4: Performance Evaluation with SPEC CPU2006 Programs on Linux Kernel.

can directly see and compare with the trusted value to detect the contamination. The rest 13 pointers point to kernel modules, and they have dynamic addresses. FPCK will compare their targeted function bodies instead of the values to decide their contamination. Note that we have acquired the source code of these rootkits, and we thus have the ground truth of the function pointers these rootkits hijacked.

There are also several interesting findings for these rootkits. In particular, two rootkits directly modify kernel function pointers (suckit via `/dev/kmem`, phalanx via `/dev/mem`), and the other seven are implemented using kernel modules. FPCK quickly identified the suspect kernel function pointers such as module `init` and `exit`. Also, because we tested these rootkits one by one, they were always inserted to the head of kernel module list. That is why the $|C|$ of `init` and `exit` functions for the rootkits are all 2, with a ptr-rexp as *(*(0xc041da18)+0x68) and *(*(0xc041da18)+0x188), respectively. Also, why its $|C|$ is 6 for `ext3_readdir`, and the reason is the derived ptr-rexp *extractor* of `ext3_readdir` is fetched from `current->fs_struct->dentry->inode ->file_operations->readdir()` (its $|C|$ is 6)

We can also notice that the majority of these rootkits tend to hijack the core kernel function pointers in global regions (such as the system call table), except `adore-2.6` which hijacks a kernel module function `ext3_readdir`. Therefore, *we believe that to hijack the kernel control flow, attackers are more in favor of the pointers that are always present in the memory such as the system call tables and are frequently used.* Again, these pointers are the persistent function pointers that FPCK aims to discover.

5.2 Performance Overhead

Extractor. Our ptr-rexp *extractor* instruments each kernel instruction to capture the pointer reference expressions. To evaluate this overhead, namely, *how slow an analyst will feel while using* FPCK *to extract the ptr-rexp*, we took SPEC CPU2006 benchmark programs and executed them atop our instrumented QEMU. The result is presented in Fig. 4. We can notice that the overhead for these programs ranges from 1.6X (for program 456.hmmer and 464.h264ref) to 10.4X (for 445.gobmk), with an average overhead of 3.3X for these benchmark programs. Overall, we can observe that for programs with system call intensive execution, they tend to have larger overhead, and these overhead mainly comes from our data dependence tracking at kernel level. Note that the instructions of user level programs will not be monitored, and our *extractor* only instruments kernel level instructions. Meanwhile, it is worth noting that *extractor* will be only executed in offline analysis.

Checker. From our design, we can notice that the major overhead of our integrity check comes from (1) identifying the function pointer location in M_u by following their ptr-rexp for all identified persistent pointers exposed by our *extractor*, (2) comparing the value or the byte stream of the targeted function code. This overhead is what a FPCK user eventually has. It turns out that this overhead is pretty small. In particular, it took less than a second (more precisely 0.10s on average) to check a 1G physical memory image in finding the hijacked pointers by each rootkit in Table 4. Note that our integrity check is not a brute force scanning of all kernel memory, and instead it is guided by a static representation of the addresses of the persistent kernel function pointers.

6. LIMITATIONS AND FUTURE WORK

Increasing the Coverage. First, like all other dynamic analysis, FPCK's results depend on the code coverage at of the running test cases, and it could miss certain execution paths and have false negatives when recognizing hijacked kernel function pointers. Though we have tried our best to combine as many test cases as possible, we have to emphasize that FPCK *itself is not a code coverage technique*. Rather, it has used the test suites to cover as many of the kernel execution paths as possible (these test case exercised paths are also often of interest to kernel rootkit developers).

Certainly, any code coverage improvement techniques will make FPCK more practical. Therefore, we plan to investigate other techniques such as symbolic execution (e.g., [7, 9]) to systematically expand and increase our coverage.

Handling Temporary Pointers. Temporary pointers include non-persistent pointers that may or may not exist, as well as the local pointers (including the function argument pointers) only reachable in certain execution context. Our ptr-rexp extractor currently does not cover these temporary pointers, and instead all of the supported pointers start from kernel global variables and they are persistent. The reason why FPCK does not check these temporary function pointers is that memory snapshot is context-less (it can be taken at arbitrary moment). Therefore, we are not able to reliably check these pointers. This also introduces problems for attackers as they cannot reliably tamper with them at arbitrary time (that is why they are more in favor of persistent pointers as discussed earlier).

One possible avenue to address this problem is to recognize the execution context (e.g., the call stack of each live kernel execution path) in the memory snapshot, and associate the context to these temporary function pointers.We leave the investigation of these techniques in our future work.

Addressing Other Attacks. FPCK currently focuses on kernel function pointer hijacking attacks exclusively and it does not handle other attacks such as the direct kernel object manipulation (DKOM). For the recent ROP based data only rootkits [30], we believe highly likely FPCK is able to detect them because these attacks still hijack the kernel function pointers (instead of the return addresses as in [19]), and also they make these pointers pointing to the existing kernel code gadget and our hash-based integrity checker will be able to spot them. We leave the evaluation of detecting these data-only rootkits also in our future work.

Other Limitations. We assumed the kernel code is immutable when designing FPCK. However, recently Linux and various BSDs began to support in-kernel just-in-time compilation, as shown in the network packet filter implementation [10]. Also, kernel tracing and interception mechanisms rely heavily on the runtime code patching, e.g., Ftrace [29], and Detours [1], which will also modify the kernel code. Certainly, FPCK will not work in these situations.

7. RELATED WORK

Since the favorite targets of many kernel attacks are the function pointers, there is a large body of research focusing on locating the kernel pointers and checking their integrity. In this section, we review and compare FPCK with these related works.

Intuitively, locating a kernel function pointer in memory would require the kernel data structure definitions. Therefore, SBCFI [24] made an early attempt of extracting the kernel data structure definitions through analyzing the kernel source code and generating a type map of kernel objects in order to identify the compromised persistent function pointers in kernel memory. Later on, various approaches have been proposed to address the limitations of SBCFI (e.g., cannot type `void` pointers, cannot prevent pointer hijacking other than the detection, and does not attempt to recognize the invariants) by systems such as Gibraltar [5], KOP [8], HookSafe [31], LiveDM [27], OSck [18], and MAS [11].

From binary analysis perspective, a number of profiling systems were proposed to understand the kernel malware behavior including the function hooking. HookFinder [34] tried to identify the hooks compromised by kernel rootkit by performing an impact analysis using dynamic tainting. HookMap [32] leveraged dynamic slicing to identify the potential hooks in the execution path used by security applications. It also required kernel symbols to annotate the hooks. Poker [28] studied the multi-aspect of kernel rootkit by traversing the type graphs extracted from the debugging information or the kernel source code. Without accessing the kernel source code, K-Tracer [21] achieved similar goals for rootkit profiling through dynamic binary analysis but required the knowledge of important kernel data structures (e.g., `EPROCESS` in Windows).

Memory analysis based approach can only detect the tampering with the kernel function pointers. To have prevention capability, HookScout [35] proposed a proactive defense by tracking the kernel heap object that contains the function pointers and preventing unauthorized modification against them. These objects are identified by tracking whether the value of a static function pointer flows to the kernel object. If so, the object must contain a kernel function pointer. The values of a function pointer were pre-identified using a static binary code analysis, facilitated by the relocation information from the kernel binary. HookScout also required the cooperation of an in-VM kernel module. In addition, it cannot be used for snapshot-based memory analysis since it does not tell how a function pointer can be reached in the kernel heap.

Most recently, HookLocator [4] demonstrated that we can periodically check the integrity of the persistent function pointers in the kernel heap pool. These heap function pointers are identified by comparing with the known function pointer values, which are pre-extracted through relocation table guided approach as HookScout, or through a cross-comparison approach between memory snapshots. While HookLocator does not require the entire kernel data structure knowledge, it still needs to know how to traverse the kernel heap pools.

Besides HookLocator, BlackSheep [6] was another lightweight approach to identify the contamination of (persistent) kernel function pointers. It did not intend to precisely identify the locations of the function pointer and then perform the integrity check, but rather directly identify them through a pattern-matching based cross comparison approach among a crowed VMs since cloud usually hosts many homogeneous VMs.

Therefore, as summarized in Table 5, we can notice that the existing efforts require either kernel source code, or debugging symbols, or relocation tables, or kernel data structure definitions to locate the kernel function pointers. There are other approaches that can directly work on binary as in HookLocator, but rely on certain heuristics to identify the kernel function pointers (such as purely based on the values) which may lead to false positives. For instance, there could be cases that a kernel heap object contains a large integer value which looks like a function pointer. In this way, HookLocator will make mistakes. As such, we still need techniques that can directly extract the locations of persistent kernel pointers (including those allocated in the dynamic kernel heap), and then inspect their integrity based on their locations. FPCK is exactly designed to achieve these goals.

Table 5: Comparison with the most related works.

Systems	wo/ Source Code	wo/ Kernel Symbols	wo/ Relocation Table	wo/ Kernel Data Structure	wo/ In-VM Assistance	wo/ Pattern Matching	Continuous Monitoring	Snapshot-based	Profiling	Detection
SBCFI [24]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓
HookFinder [34]	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗
HookMap [32]	✓	✗	✓	✓	✓	✓	✓	✗	✓	✗
Gibraltar [5]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓
K-Tracer [21]	✓	✗	✓	✗	✓	✓	✓	✗	✓	✗
Poker [28]	✗	✗	✓	✗	✓	✓	✗	✓	✓	✗
KOP [8]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓
HookSafe [31]	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓
HookScout [35]	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓
LiveDM [27]	✗	✗	✓	✗	✓	✓	✓	✗	✓	✓
OSck [18]	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓
HUKO [33]	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓
MAS [11]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓
BlackSheep [6]	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓
HookLocator [4]	✓	✓	✗	✗	✓	✗	✗	✓	✓	✓
FPCK	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓

8. CONCLUSION

We have presented FPCK, a binary exclusive approach for automatically locating kernel function pointers for their integrity checking. To locate the function pointers, we propose an on-demand *pointer reference expression* generation algorithm that extracts data reference expressions for kernel pointers during the kernel execution. This *pointer reference expression* encodes how a pointer is accessed through the combination of a base address (usually a kernel global variable) with certain offset and further pointer dereference operations, and allows us to locate them in different instances of a kernel memory. To check their integrity, we propose to compare the function pointers of an untrusted kernel memory image against the ones in the trusted kernel. We have implemented a prototype of FPCK. Our experimental results with a number of real world kernel malware form Linux platform show that FPCK can automatically identify all the hijacked pointers for these testing samples within just few seconds. We believe FPCK will be particularly useful for both public and private cloud providers to check the guest kernel integrity and estimate the function pointer damages inflicted by kernel malware.

9. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback. 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.

10. REFERENCES

[1] Detours. https://research.microsoft.com/en-us/projects/detours/.

[2] Linux test project. http://ltp.sourceforge.net/.

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

[4] AHMED, I., RICHARD, G., ZORANIC, A., AND ROUSSEV, V. Integrity checking of function pointers in kernel pools via virtual machine introspection. In *Proc. of th 16th Information Security Conference (ISC'13)* (Dallas, Texas, USA, 2013).

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

[6] BIANCHI, A., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Blacksheep: detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security(CCS'12)* (Raleigh, NC, USA, 2012).

[7] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, 2008).

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

[9] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems(ASPLOS'11)* (Newport Beach, California, USA, 2011), pp. 265–278.

[10] CORBET, J. A jit for packet filters, 2011. http://lwn.net/Articles/437981/.

[11] CUI, W., PEINADO, M., XU, Z., AND CHAN, E. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of the 21st conference on USENIX Security Symposium(Security'12)* (Bellevue, WA, USA, Aug. 2012).

[12] 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 the 2012 IEEE Symposium on Security and Privacy(S&P'12)* (San Francisco, CA, May 2012).

[13] FU, Y., AND LIN, Z. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments(VEE'13)* (Houston, TX, March 2013).

[14] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows xp kernel crash analysis. In *Proceedings of the 20th Conference on Large Installation System Administration(LISA'06)* (Washington, DC, 2006), USENIX Association, pp. 12–12.

[15] GU, W., KALBARCZYK, Z., IYER, K., AND YANG, Z. Characterization of linux kernel behavior under errors. In *Proceedings of 2003 International Conference on Dependable Systems and Networks(DSN'03)* (San Francisco, CA, USA, 2003), pp. 459–468.

[16] GU, Y., FU, Y., PRAKASH, A., LIN, Z., AND YIN, H. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)* (San Jose, CA, USA, 2012).

[17] GU, Y., FU, Y., PRAKASH, A., LIN, Z., AND YIN, H. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing* (2014).

[18] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems(ASPLOS'11)* (Newport Beach, California, USA, 2011), ACM, pp. 279–290.

[19] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium(Security'09)* (Montreal, Canada, 2009), pp. 383–398.

[20] JOHNSON, R., AND WAGNER, D. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium(Security'04)* (San Diego, CA, USA, 2004), pp. 9–9.

[21] LANZI, A., SHARIF, M. I., AND LEE, W. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 2009 Network and Distributed System Security Symposium(NDSS'09)* (San Diego, CA, USA, 2009).

[22] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND 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'11)* (San Diego, CA, February 2011).

[23] 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).

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

[25] QUYNH, N. A. Operating system fingerprinting for virtual machines, 2010. In DEFCON 18.

[26] RHEE, J., LIN, Z., AND XU, D. Characterizing kernel malware behavior with kernel data access patterns. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security(AsiaCCS'11)* (Hong Kong, March 2011).

[27] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection (RAID 2010)* (Ottawa, Canada, September 2010).

[28] RILEY, R., JIANG, X., AND XU, D. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)* (Nuremberg, Germany, 2009), pp. 47–60.

[29] ROSTEDT, S. Debugging the kernel using ftrace, 2009. http://lwn.net/Articles/365835/.

[30] VOGL, S., PFOH, J., KITTEL, T., AND ECKERT, C. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)* (San Diego, CA, USA, 2014).

[31] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security(CCS'09)* (Chicago, Illinois, USA, 2009), pp. 545–554.

[32] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection(RAID'08)* (Cambridge, MA, USA, 2008), pp. 21–38.

[33] XIONG, X., TIAN, D., AND LIU, P. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)* (San Diego, CA, 2011).

[34] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium(NDSS'08)* (San Diego, CA, USA, 2008).

[35] YIN, H., POOSANKAM, P., HANNA, S., AND SONG, D. HookScout: Proactive binary-centric hook detection. In *Proceedings of Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)* (Bonn, Germany, July 2010).