Multi-Aspect, Robust, and Memory Exclusive Guest OS Fingerprinting

Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, *Member, IEEE*, and Heng Yin, *Member, IEEE*

Abstract—Precise fingerprinting of an operating system (OS) is critical to many security and forensics applications in the cloud, such as virtual machine (VM) introspection, penetration testing, guest OS administration, kernel dump analysis, and memory forensics. The existing OS fingerprinting techniques primarily inspect network packets or CPU states, and they all fall short in precision and usability. As the physical memory of a VM always exists in all these applications, in this article, we present OS-SOMMELIER$^+$, a multi-aspect, memory exclusive approach for precise and robust guest OS fingerprinting in the cloud. It works as follows: given a physical memory dump of a guest OS, OS-SOMMELIER$^+$ first uses a code hash based approach from kernel code aspect to determine the guest OS version. If code hash approach fails, OS-SOMMELIER$^+$ then uses a kernel data signature based approach from kernel data aspect to determine the version. We have implemented a prototype system, and tested it with a number of Linux kernels. Our evaluation results show that the code hash approach is faster but can only fingerprint the known kernels, and data signature approach complements the code signature approach and can fingerprint even unknown kernels.

Index Terms—Operating system fingerprinting, virtual machine introspection, memory forensics

1 INTRODUCTION

Operating system (OS) fingerprinting aims to identify the exact version of the OS running on a target machine. Such information is extremely useful for many applications, such as penetration testing, system administration (e.g., kernel updates), virtual machine management, and digital forensics. For example, with knowledge of the exact OS version, we can launch various targeted probes and attacks for the purpose of penetration testing. It is also helpful in administration, as administrators can often perform regular OS fingerprinting to keep their OS inventory clean and updated with the most recent patches.

Meanwhile, with the rapid deployment of cloud computing, we are facing an increasing need for fingerprinting the guest OS for virtual machine (VM) management in the cloud, especially for IaaS cloud providers. One typical example for cloud management and security is the virtual machine introspection (VMI) [17]. To obtain the semantic-level view of the VM and interpret the system states and events [8], [13], [22], [29], we often have to know in advance the *precise* kernel version of the guest OS (such that we can use the corresponding kernel data structure to interpret the kernel memory). However, the information of kernel version may not be immediately available to the cloud provider or, even if available, may not be accurate.

To alleviate the manual effort involved in bridging the semantic gap in VMI, we recently have developed a set of dual-VM, binary code reuse based systems (e.g., [13], [14], [15], [31]) to automatically bridge the semantic gap. These systems rely on the corresponding trusted kernel installed in a secure VM and perform an on-line kernel data redirection to introspect the guest VM. That is, without the precise guest OS kernel information, all of them would not work. Also, we have to perform precise OS kernel fingerprinting, as any minor differences between the two kernels will stop these systems.

In addition, the knowledge of the precise OS version is also crucial to many other applications such as memory forensic analysis, which aims to collect evidence of digital crimes from a physical memory dump of a live system. Similar to VMI, the memory forensic tools (e.g., [38]) often have to know the precise OS version in advance in order to parse the memory dump correctly. This knowledge may not be immediately available, or may be tampered by the attackers to thwart forensic analysis.

Unfortunately, the existing OS fingerprinting techniques fall short in *precision* and *usability*. More specifically, network-based fingerprinting (e.g., nmap [16], [18] and Xprobe2 [4]) recognizes the discrepancies in network protocol implementations by sending crafted packets and analyzing the differences in the responses. This network-based approach is often imprecise and cannot pinpoint the minor OS differences such as Linux-2.6.35 versus Linux-2.6.36. Moreover, the network-based approach becomes less usable since modern OSes (e.g., Windows 7) disable many network services by default.

Two other recent systems explore the host based information such as CPU register values [30] and the interrupt handler code hashes [10] for guest OS fingerprinting. They are still not precise enough to pinpoint the different service packs for Windows and minor revisions for Linux and BSD

- *Y. Gu, Y. Fu, and Z. Lin are with the Department of Computer Science, The University of Texas at Dallas, 800 W. Campbell RD, Richardson, TX 75080. E-mail: {yufei.gu, yangchun.fu, zhiqiang.lin}@utdallas.edu.*
- *A. Prakash and H. Yin are with the Department of Computer Science Syracuse University, 400 Ostrom Avenue, Syracuse, NY 13210. E-mail: {arprakas, heyin}@syr.edu.*

Manuscript received 15 Dec. 2013; revised 14 May 2014; accepted 4 June 2014. Date of publication 10 July 2014; date of current version 30 Jan. 2015. Recommended for acceptance by D.S.L. Wei, S. Pearson, K. Matsuura, P.P.C. Lee, and K. Naik.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2014.2338305

families. File system based fingerprinting is another approach. Tools like virt-inspector [23] examine the file system of the VM, look for main kernel code, and determine the OS version. This approach is straightforward, but is not feasible for a VM with encrypted file system, resulting from privacy concerns of cloud customers.

Therefore, to provide a strong support for security and forensics tasks in the cloud, we need to revisit the OS fingerprinting problem. Since the memory state of a VM is always available to the cloud provider, we propose to take a memory exclusive approach for OS fingerprinting. This new approach needs to be *precise* enough to recognize the minor versions of an OS kernel, and *robust* enough to counter the evasions. To make this technique as generic as possible, we choose not to rely on other inputs. As a result, our memory exclusive fingerprinting can provide a direct support to many other applications such as memory forensics and kernel dump analysis.

To answer these needs, we have developed a new, multi-aspect, memory exclusive guest OS fingerprinting system, called OS-SOMMELIER$^+$, which is an extension of our prior system OS-SOMMELIER [19]. It works as follows: given a physical memory dump of a guest OS, OS-SOMMELIER$^+$ first uses a code hash based approach from kernel code aspect to determine the guest OS version. If code hash approach fails, OS-SOMMELIER$^+$ then uses a kernel data signature based approach from kernel data aspect to determine the version. The reason to have both approaches is that code signature approach is fast, but too sensitive (any recompilation of the same kernel might lead to different hashes); data signature approach is vice-versa, namely, it is agnostic to the kernel code changes (for the same version), but too slow.

However, it is non-trivial to compute the kernel code hash given only the physical memory of a guest OS, especially for the widely used ×86 architecture in the cloud. For instance, how to distinguish the main kernel code from the rest of code and data, and how to tolerate real-world issues such as address space layout randomization (ASLR [6], [7], [37], [39]), and the dynamic kernel code patches (i.e., hot-patches [35]). It is also non-trivial to compute robust kernel data signatures. For instance, given the significant amount of kernel data structures, how many we should use to efficiently fingerprint a kernel, and what kind of robust signatures we should derive.

To address these challenges, we have devised a suite of techniques including *core kernel code identification*, *correlative disassembling*, and *normalized signature matching* in our code signature approach, and *point-to invariant* (or *structural invariant*), *loop invariant*, and *size invariant* in our data signature approach. Our experimental results with 27 Linux kernels show that our code hash based approach can quickly fingerprint all the known OSes we tested without any false positives and false negatives, but when recompiling the same OS with different flags, it cannot recognize them. In contrast, data signature based approach is able to infer those unknown OSes but it runs too slow (two order of magnitude slower).

The main contribution of this paper is highlighted as follows:

- We present a multi-aspect approach to precisely fingerprint an OS kernel when provided with *only* a physical memory dump. Our approach is general without relying on any heuristics for particular OSes, and it uniformly works for all the kernels we tested.
- We devise a set of novel techniques to automatically identify *core* kernel code in the physical memory to compute kernel code hash signatures, by exploring the direct control flow transfer pattern in kernel code and the unique kernel code (i.e., system level) instructions, and normalize the kernel code pages by retaining the op-code and register operand of the disassembled instructions and hashing them as the signatures.
- We also devise an array of novel techniques to automatically compute robust kernel data signatures, by exploring the point-to relation between data structures, and the strong loop invariant and size invariant among them.
- We have implemented our system OS-SOMMELIER$^+$, and tested it with over a variety of recent Linux kernels. Our experimental results show that the code aspect approach is very precise and fast to the known kernels but too sensitive, whereas data aspect approach is less sensitive but slow. These two can be combined together to offer a best result for memory exclusive OS fingerprinting.

2 BACKGROUND AND OVERVIEW

2.1 Problem Statement

Given a memory snapshot of a running VM in the cloud, we aim to precisely determine its OS version. In particular, we have three major design goals: *precision*, *efficiency* and *robustness*.

- *Precision.* We need to determine the OS family and the exact version. For instance, given a Linux kernel, not only we need to know its major version (e.g., 2.6 or 3.0) but also its minor version. This is because many security tasks (e.g., [13], [22], [29]) have to rely on the exact OS version to make OS-specific decisions.
- *Efficiency.* Given that the cloud provider usually manages a large volume of live VMs, it becomes necessary to obtain the information of the OS version within a short period of time for each VM.
- *Robustness*. A VM running in the cloud may have been compromised and attackers may manipulate the memory state of the VM to mislead the fingerprinting system. Thus, our technique needs to be robust to counter various attacks, or at least raise a significant high bar for adversaries.

Threat model and our assumption. We assume the integrity of the main kernel code. Recent advance in trusted computing techniques (e.g., [27], [33]) and virtual machine security (e.g., [5], [9], [41]) can easily ensure the integrity of the kernel code pages. For this reason, we focus on the main kernel code to achieve robustness in memory-based OS fingerprinting. Consequently, since the main kernel code is not modified, attackers cannot modify (e.g., shuffling the pointer fields) the main data structure as well. Otherwise, when trusted kernel dereference certain pointer fields, it might lead to crashes.

2.2 Challenges and Key Techniques

A program in general contains both code and data at runtime, and there is no exception for an OS kernel. As such, we propose to fingerprint an OS kernel from both code and data perspectives. More specifically, we propose computing core kernel code hashes as code signatures (Section 2.2.1), and deriving data structure invariants for data signatures (Section 2.2.2). In the following, we describe the challenges we encountered and briefly sketch our solutions.

2.2.1 Code Signature Based Approach

To compute code signature, our key idea is to correctly identify the core kernel code from a physical memory dump, and then calculate hashes to precisely fingerprint an OS. To realize this idea, we have to address the following challenges:

Correctly disassembling the kernel code. To compute hash values of the kernel code in memory, we must correctly disassemble it. Unfortunately, it is widely known that correct code disassembly is still an open problem for $\times 86$ architecture. There are two main reasons: (i) it is common to have code and data interleaved, and it is hence hard to differentiate code and data; and (ii) because $\times 86$ instructions have varied lengths, a completely different instruction sequence will be disassembled if starting from a wrong instruction location.

To address this challenge, we propose a novel *correlative disassembling* technique by leveraging the correlation between a call instruction and the function prologue of the call target. More specifically, we believe a location to be a function entry point if and only if: (i) a function prologue exists starting from this location; and (ii) a call instruction is found, where this location is exactly the call target. Based on this correlation, we are confident that the identified function body is truly a function. However, we may not be able to identify all the kernel code. Some functions may not have well-defined function prologues. Fortunately, we can accept this situation, as there is a large amount of kernel code and the correctly identified portion is sufficient enough to serve the purpose of our fingerprinting.

Differentiating the main kernel code from the rest of code and data. The kernel code includes the code of the main kernel, as well as the code of the device drivers (i.e., kernel modules). We aim to compute the hashes for the main kernel code only, because the presence of the other kernel modules is mainly determined by the hardware configuration of a system. Two systems may have the same OS version installed, but due to different hardware configurations, they may have completely different sets of kernel modules.

To this end, we propose a *direct call based clustering technique*, to group identified function bodies into clusters, each of which is either the main kernel code or the code for a kernel module. Our insight is that the target of a direct function call has to be located in the same code module as the direct function call itself. This is because the target of a direct function call is determined at compilation time, the call target and the call site must be present in the same module. Based on this insight, we cluster the disassembled code into code modules.

Then, to tell which code module is the main kernel, we have another insight: certain instructions have to appear in the main kernel to implement some important functionality (such as context switch and cache flushing), and it is unlikely for the other kernel modules to have these instructions.

2.2.2 Data Signature Based Approach

Unlike kernel code that tends to have smaller size, there are huge amount of kernel data. To derive data signatures for OS fingerprinting, we have to address the following challenges:

Selecting the robust data signatures. Intuitively, we might need to select those data with constant values as signatures. For instance, if we find certain strings exclusively exist in certain versions of an OS, these strings would be ideal candidates. However, it would be too easy to generate bogus strings to evade or mislead this fingerprinting. Therefore, we have to look for robust signatures.

Among all the data inside memory, we notice that it is challenging to modify the values of pointers because of the constraints from the pointer shapes. In particular, pointers usually have specific types, changing a pointer value may lead to change the target address type, and thus rendering kernel crashes. It is also harder to be evaded than purely value-based approach. For example, one has to change the entire points-to shape, including the data structure shape of the corresponding target address. Moreover, adversary also has to change all the data structure instances; otherwise we can still find the true instance to fingerprint the kernel. In contrast, in value-based approach, one only needs to change a single value. As such, we propose exploring the point-to constraints, or *structural invariant* as the signatures.

Also, the size of a data structure is also harder to be forged than purely values, as adversary has to spent significant more efforts. For instance, to forge a `task_struct`, adversary has to allocate a real `task_struct` and allocate other necessary data structures to evade the *structural invariant*. Therefore, *size invariant* can be considered as another orthogonal dimension of a data signature. Intuitively, when more signatures are used to fingerprint an OS, it will be more precise and robust.

Minimizing signatures to improve efficiency. An OS kernel usually has tens of thousands of data structures. An ideal case would be to build the entire data structure graph constrained by the point-to relation as the signature. However, this would be inefficient given the huge amount of kernel data. Therefore, we have to minimize the size of the *structural invariant*.

Fortunately, we have a new observation that there often exists a special *structural invariant* called *loop invariant* enforced by the pointer fields among data structures, and such a *loop invariant* can be well-modeled for data fingerprinting and can significantly reduce the number of checks when validating a *structural invariant*. In particular, this loop invariant is a loop-shaped data reference path starting from a given memory address x and reaching back x through pointer dereferencing. In addition, such a loop invariant widely exists in an OS kernel, and it is non-trivial to be modified as pointers are hard to be

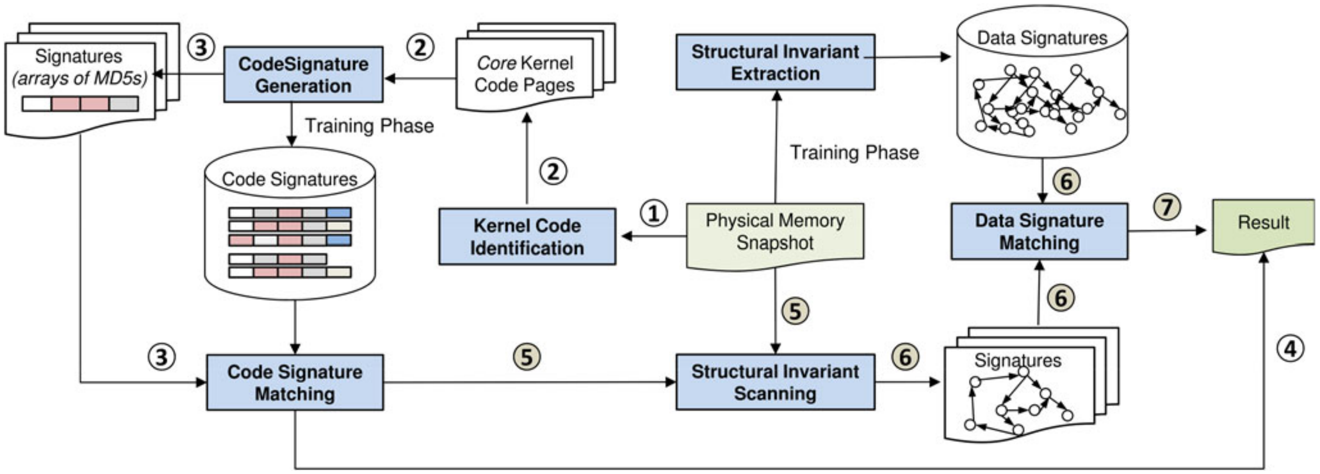

Fig. 1. Overview of OS-Sommelier$^+$.

manipulated. For example, in Linux kernel, if `A` is an instance of `task_struct`, then `(A->thread_info)->task==A`. Other interesting cases include a circular list or a doubly-linked list which has pointer to point back. More formally, it could be defined as a fixed pointer function f where $x = f(x)$ because of the convergence, and our goal is to automatically identify the unique f which can fingerprint a particular OS.

2.3 System Overview

An overview of OS-Sommelier$^+$ is presented in Fig. 1. There are six key components in our system: (1) *Kernel Code Identification*, (2) *Code Signature Generation*, (3) *Code Signature Matching*, (4) *Structural Invariant Extraction*, (5) *Structural Invariant Scanning*, and (6) *Data Signature Matching*. The first three components belong to code signature based approach, and the last three belong to data signature based approach.

More specifically, given a physical memory snapshot (**Step** ①), OS-Sommelier$^+$ first invokes the *Kernel Code Identification* to traverse the kernel page tables and identify only the kernel code pages (based on the page directory entry and page table entry bit properties), and further it splits the kernel code based on the virtual addresses and the internal caller-callee relation to identify the "*core*" kernel code pages.

Next (**Step** ②), *Code Signature Generation* will use a correlative disassembling technique to neutralize the side effect of ASLR, and then hash (MD5) each disassembled page and store it in an array based on the normalized virtual address of each hashed page. **Step** ① and **Step** ② can also be used in the training phase to generate the ground truth MD5 of the guest OS. Then (**Step** ③), *Code Signature Matching* adopts a string matching algorithm to compare the MD5-array with a database that contains the array-signatures for all the possible OSes. If the signature matches, it directly output the result (**Step** ④).

Otherwise (**Step** ⑤), OS-Sommelier$^+$ then looks into the data aspect of the OS kernel, and invokes *Structural Invariant Scanning* to scan, and match (**Step** ⑥) with the signatures produced in the training phase by *Structural Invariant Extraction* component, and output (**Step** ⑦) the final results (either matched with a known OS with specific version, or returns unknown). The *structural invariants*, *size invariants*, and *loop invariants* are generated by *Structural Invariant Extraction* component.

To simplify the paper presentation, we focus our discussion on the widely used 32-bit ×86 architecture, running Linux kernel. However, we believe our techniques can be generalized and adapted to support other CPU architectures, and other OSes (e.g., Windows, FreeBSD and Minix).

3 Code Signature Based Approach

3.1 Kernel Code Identification

OS kernel is composed with core kernel code and kernel modules. Because a kernel module can exist in multiple different kernels, we have to exclude the kernel module code in our code hash computation. Given a physical memory snapshot, there are three steps to reach the core kernel code: (**Step-I**) we will perform the virtual to physical (`V2P`) address translation for kernel space by checking each Page Directory Entry (PDE), whose base address is pointed by control register CR3, and Page Table Entry (PTE), and group them based on the page table properties to a number of clusters; (**Step-II**) we will further search from the clusters to identify the possible kernel code by searching the special kernel instruction sequences; and (**Step-III**) finally we will further narrow down the kernel code. The first two steps are used to identify the possible kernel code, and the third step is to identify the "*core*" kernel code from the kernel code identified in **Step-II**.

Step-I: Searching possible kernel code and clustering. With the Page Global Directory (PGD) identification approach we developed in OS-Sommelier [19], we first identify PGDs from a given memory snapshot, from which to find the entire kernel space by checking the system bit in the page table entries. We group these kernel pages into clusters, based on the PDE and PTE properties. In particular, we put contiguous pages into one cluster, if these pages share the identical page properties. The output of this step is a number of clusters, denoted as C_K, and each cluster (C_{Ki}) contains the pages whose virtual addresses have been resolved and these pages share the identical PTE bits and the page size bit in PDE.

Step-II: Identifying the possible kernel code. Next, we aim to identify which cluster contains the main kernel code. One

TABLE 1
X86 System Instruction Distributions in Kernel Code Pages

System	Inst.	Linux-2.6.32		Windows-XP		FreeBSD-9.0		OpenBSD-5.1		NetBSD-5.1.2		Minix-3.2.1	
Instructions	Length	#Inst.	#pages	#Inst.	#pages	#Inst.	#pages	#Inst.	#pages	#Inst.	#pages	#Inst.	#pages
LLDT	3	17	10	4	3	5	3	5	4	2	2	2	2
SLDT	3	1	1	1	1	1	1	2	2	1	1	0	0
LGDT	3	10	8	1	1	1	1	3	2	3	2	2	2
SGDT	3	4	4	5	4	1	1	2	2	1	1	2	2
LTR	3	2	2	2	2	6	5	5	3	2	2	2	2
STR	3	2	2	2	2	1	1	1	1	2	2	0	0
LIDT	3	7	6	2	2	5	4	5	3	2	2	4	2
SIDT	3	2	2	5	4	1	1	2	2	1	1	2	2
MOV CR0	3	68	16	65	21	33	8	45	12	14	5	4	2
MOV CR2	3	5	5	2	2	2	2	12	5	5	2	6	2
MOV CR3	3	70	18	24	10	49	12	17	6	16	7	14	2
MOV CR4	3	94	23	22	7	25	7	24	8	12	5	4	2
SMSW	4	0	0	0	0	5	1	0	0	0	0	0	0
LMSW	3	0	0	0	0	5	1	0	0	0	0	0	0
CLTS	2	6	5	3	1	6	1	7	2	1	1	2	2
MOV DRn	3	0	0	262	8	0	0	0	0	0	0	0	0
INVD	2	0	0	0	0	5	1	2	1	2	1	0	0
WBINVD	2	28	14	6	3	15	8	14	8	1	1	0	0
INVLPG	3	7	3	4	3	24	10	14	4	3	2	2	2
HLT	1	12	6	1	1	5	5	4	1	4	3	2	2
RSM	2	0	0	0	0	0	0	0	0	0	0	0	0
RDMSR3	2	113	25	1	1	76	17	79	16	2	1	2	2
WRMSR3	2	111	28	1	1	51	15	54	17	2	1	2	2
RDPMC4	2	0	0	0	0	0	0	1	1	1	1	0	0
RDTSC3	2	26	12	21	7	14	4	5	3	3	2	2	1
RDTSCP7	3	0	0	0	0	0	0	0	0	0	0	0	0
XGETBV	3	0	0	0	0	0	0	0	0	0	0	0	0
XSETBV	3	3	3	0	0	0	0	0	0	0	0	0	0

possible solution would be to search for the page which contains instructions based on the code distributions [12]. However, this approach tends to be computation-intensive, contradicting our design goal of efficiency. Then an intuitive approach would be to search for the special system instruction sequences that (1) often appear in main kernel code, (2) have unique pattern (short sequence will have false positive), and (3) not in kernel modules.

According to the ×86 instruction set [21], there are in total 28 system instructions and their instruction length and distributions in Linux, Windows, BSD UNIX, and Minix are summarized Table 1. We could observe that not all of them can be used as the searching sequence, such as SLDT and RDMSR3 as they only appear few times and are distributed in few pages. Therefore, eventually, we decide to choose MOV CR3 instruction (as it is related to PGD update and process context switch). Moreover, a closer investigation with this instruction yields the following 6 bytes unique instruction sequence:

```
0f 20 d8: mov EAX, CR3;
0f 22 d8: mov CR3, EAX;
```

In fact, these two consecutive instructions are used by the modern OS to force a TLB flush, avoiding problems related to implicit caching [21]. We confirm that this sequence (`0f 20 d8 0f 22 d8`) appears in all the kernels we tested.

Very often (as shown in Table 3), the output of **Step-II** is a single cluster for the main kernel code. Considering that there may still exist some noise, we accept the possibility that the output may be a small number of clusters, denoted as C_{Kk}.

Step-III: Narrowing down the core kernel code. In these possible kernel code clusters, only one cluster has the core kernel code. Such a cluster may still include the code for the other kernel modules and the data, due to the imprecision in previous steps. To precisely narrow down the core kernel code, we leverage an insight from direct function calls.

More specifically, as the target of a direct function call has been resolved at compilation time, this target is either within the same code module, or from a static library. For an OS kernel whose main kernel code can be relocated, any function call between the main kernel and a device driver must be indirect, because the call target cannot be determined at compilation time. That is, the direct call instruction and the call target have to be located in the same code module. Therefore, by checking direct calls, we can further narrow down the core kernel code.

The situation is more complicated when the main kernel code is static (e.g., Linux). In this case, a device driver (i.e., kernel module) can invoke a direct function call (e.g., `printk`) into the main kernel, as its target address (i.e., the entry point of `printk`) can be easily resolved (and this target address is dynamically patched by the driver loader). Nevertheless, the main kernel cannot make a direct function call into a device driver, because its address can only be determined at load time. To identify the main kernel code in this case, we rely on another insight: *the main kernel always occupies the lowest kernel space, leaving the higher kernel space*

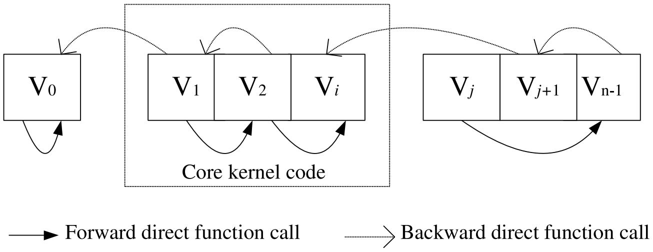

Fig. 2. Illustration of core kernel code clustering.

for the device drivers. This is because the OS cannot predict how many drivers will be loaded and how much kernel space is to be allocated for the drivers. In our experiments, this observation holds true for all the kernels whose main kernel code are static. Therefore, we propose to explore the *direct forward function call* relation.

A direct forward function call is a call instruction whose operand is a positive value (e.g., the case for `e8 2a 25 38 00`), and a direct backward function call is a call instruction whose operand is a negative value (e.g., `e8 2a 25 38 ff` where `2a 25 38 ff` is a negative value). As the main kernel occupies the lowest kernel space, a direct forward function call has to be within the same code module. Thus, as illustrated in Fig. 2, by searching direct forward calls, we can exclude the device drivers. To identify and verify the existence of a direct call instruction, we use the *correlative disassembling* approach, which will be discussed in the next subsection (Section 3.2).

Our detailed clustering algorithm is presented in Algorithm 1. For each page in C_{Kk}, we will check whether there is a direct forward function call (line 9), and if so, we will update the ending boundary ($C.end$) for the current cluster (line 11). We will also check whether the current cluster ends (line 14), if so we will store the current cluster to a temporary cluster set T (line 15), and allocate a new cluster (lines 16-19).

Finally, we will search for the largest size cluster which contains the 6-byte instruction sequence for TLB flushing, and we identify this cluster to be the main kernel code.

3.2 Code Signature Generation

An intuitive signature generation scheme is to hash (MD5) each page in C_{CK} as a signature. This approach would work if there is no code relocation, or kernel ASLR. However, such an approach will fail for some modern OSes such as the recent Windows-7, which actually randomizes the kernel instruction addresses, and during the randomization some code and data labels for some instructions are changed.

Thus, we have to neutralize the randomization. To this end, we introduce a *code normalization* that distills the memory and immediate operands and only hashes the opcode and register operand, based on a robust disassembly. In addition, we also observe that the operand for a direct call instruction remains identical, because the target is referenced as a relative offset. That explains why the direct forward call based clustering is general enough to deal with the randomized kernels.

Correlative disassembling. Robust disassembling in general is a challenging task in ×86, since code and data could be mixed, and code could start at arbitrary location (may not be aligned). In OS-SOMMELIER$^+$, we take a special and robust approach by exploring the constraint from the direct call instruction and its targeted function prologue.

```
0xc1087c86: e8 25 2c 00 00    call 0xc108a8b0
...
0xc108a8b0: 55                push ebp
0xc108a8b1: 89 e5             mov ebp, esp
```

Fig. 3. Exploring the constraint between a caller instruction and the callee prologue for robust disassembling.

Algorithm 1. Core Kernel Code Identification

```
Require: Vaddr($t$) returns the virtual address for $t$. Page ($t$) returns the page in which virtual address $t$ resides.
FunTarget($i$) returns the target address for
        instruction $i$.
Input: The kernel code cluster $C_{Kk}$ that contains a
        number of pages whose virtual address has
        been resolved;
Output: $C_{CK}$, which is a subcluster of $C_{Kk}$ that contains
        the core kernel code part.
 1: CoreKernelCodeIdentification($C_{Kk}$){
 2: $C \leftarrow$ new $Cluster$
 3: $C.start \leftarrow$ Vaddr($p_0$)
 4: $C.end \leftarrow$ Vaddr($p_0$) $+ 4096$
 5: $C.page \leftarrow \emptyset$
 6: $T \leftarrow \emptyset$
 7: for each $p_i \in C_{Kk}$ do
 8:   $C.page \leftarrow C.page \cup \{p_i\}$
 9:   for each DirectForwardFunCall $f \in p_i$ do
10:     if FunTarget($f$) $> C.end$ then
11:       $C.end \leftarrow$ FunTarget($f$)
12:     end if
13:   end for
14: if Vaddr($p_i$) $+4096 > C.end$ then
15:       $T \leftarrow T \cup C$
16:       $C \leftarrow$new $Cluster$
17:       $C.start \leftarrow$Vaddr($p_i$)
18:       $C.end \leftarrow$Vaddr($p_i$) $+ 4096$
19:       $C.page \leftarrow \emptyset$
20:     end if
21: end for
22: $T \leftarrow T \cup C$
23: $C_{CK} \leftarrow T[0]$
24: for each $c \in T$ do
25:   if (TlbFlush $\in c$ and $|c| > |C_{CK}|$) then
26:     $C_{CK} \leftarrow c$
27:   end if
28: end for
29: return $C_{CK}$
30: }
```

More specifically, considering a direct call instruction `call0xc108a8b0` (with the machine code `e8 25 2c 00 00`) shown in Fig. 3 as an example, the operand of this instruction `25 2c 00 00` ($0 \times 2c25$) is the displacement to the target callee address ($0 \times c108a8b0$), which can be computed from the PC of the direct call instruction (0xc1087c86) plus the displacement (0x2c25) and the instruction length (5). Meanwhile, the function prologue of the callee instruction also has a unique pattern, namely, with a machine code `55 89 e5` which is the instruction sequence of `push ebp, move ebp, esp`. As a result, by searching for machine code `e8 x x x x` and computing its callee target address, as long as the targeted callee address has the pattern of a function prologue, we will start to disassemble the target page from the callee prologue. When encountering a `ret` or a direct or an indirect `jmp` instruction, we stop disassembling this function. In other words, our disassembling adopts a linear sweep algorithm [32]. We have tested this correlative disassembly approach with many binary programs including the OS kernel (to be presented in Section 5) and user level binary (c.f., [20]), and we did not encounter any false positives.

We devise this disassembling algorithm specially for our fingerprinting: (1) it is simple and efficient; and (2) the dissembled instructions are correct with high confidence. These benefits are achieved at the cost of a lower coverage. For a better coverage, we could have taken a recursive disassembling approach (e.g., [25]). We do not choose this sophisticated approach because the disassembly coverage is not a crucial factor for OS fingerprinting and it would incur significant performance impact.

Algorithm 2. Code Signature Generation

Require: **`LinearSweepDisass`**(p) returns one or more page that contains disassembled code which has distilled the memory operand and un-disassembled code with 0. **`Vaddr`**(t) returns the virtual address for t. **`FunTarget`** (f) returns the target address for function f. **`Prologue`** (t) returns true if the virtual address starting at t is a function prologue. **`WinthinPage`**(p,q) returns whether p and q are within the same page. **`MD5`**(d) returns the hash value of d.

Input: The core kernel code cluster C_{CK} that contains a number of pages whose virtual addresses have been resolved;

Output: A signature array S in which each element is a MD5 for the disassembled page.

```
1:  SignatureGeneration($C_{CK}$){
2:  for each $p_i \in C_{CK}$ do
3:      $p_i.dis \leftarrow \emptyset$
4:      for each DirectFunCall $f$ in $p_i$ do
5:          $taddr \leftarrow$ FunTarget($f$)
6:          if WithinPage($taddr$, $f$) and Prologue($taddr$)
            then
7:              $p_i.dis \leftarrow p_i.dis \cup \{taddr$ mod $4096\}$
8:          end if
9:      end for
10:     $data \leftarrow$ LinearSweepDisass($p_i$)
11:     $index \leftarrow$ Vaddr($p_i$)-Vaddr($p_0$)
12:     $S[index] \leftarrow$MD5($data$))
13: end for
14: return $S$ }
```

The detailed signature generation algorithm is presented in Algorithm 2. In particular, for all the pages whose virtual addresses have been resolved in C_{CK}, we first search all the possible starting addresses for disassembly (because within one page there could be multiple function prologues) in each page by verifying the direct function call instruction and its prologue (lines 5-8). After we have identified all the starting addresses for disassembly in a page, we next disassemble it (line 10). Our linear sweep disassembler will stop when we encounter a `ret` or a direct or indirect `jmp` instruction. For the rest of un-disassembled code and data, we will clear it with 0. We will also zero out the memory operands and immediate operands of the disassembled instructions. Eventually, we will have a new page $data$ (or more than one if the disassembling end point is in the other pages) which contains all the opcode of the disassembled instructions and some of its operands such as registers.

Note that we could have generated just one MD5 for the entire data in S. But to support a sensitive detection of any kernel code modification, and the tolerance of possible kernel page swap, we introduce a *signature normalization* technique. Our *signature normalization* will order the MD5-signatures from each disassembled page and store them in an array indexed on the normalized virtual address of the page (lines 11-12). Thus, for the swapped page, only the array element indexed by that particular missing page will not have a hit in the signature matching. Meanwhile, our experiment with the real world OSes shows that our final array-organized signatures are extremely strong and only a few hash values are unique enough to precisely fingerprint an OS.

3.3 Code Signature Matching

As a final step, to identify the precise OS version, the signature of the memory snapshot is "string" matched against the database of signatures where the "string" is composed with 32-bytes MD5 values. Here, we use the standard KMP [24] string matching algorithm. The slightly tweak is that we have to represent the original single character of a string with a 32-bytes MD5 value. For the comparison of the 32-bytes MD5 value, we just sequentially compare each byte since the hash function tends to have normal distributions for each character. The details are elided (as the KMP string matching is a standard algorithm).

4 Data Signature Based Approach

Code signature based approach is very sensitive to the kernel code: even a recompilation of the same kernel with different compilation option might lead to different signatures. To tolerate such sensitivity, we need to explore other aspect of OS kernel to fingerprint the OS kernel. In this section, we first present our observation with data signatures in Section 4.1, and then present the detailed design of our techniques from Sections 4.2 to 4.3.

4.1 Key Observation

As we have discussed in Section 2.2.2 that we eventually decide to use pointer values, especially the loop invariant, a special case of structural invariant (exhibited by kernel data structure when following pointer dereference) as one of the data signatures. The fundamental reason of why we can use the loop invariant is that (1) they widely exist, (2) they are unique, and (3) they are self-certifying. More specifically.

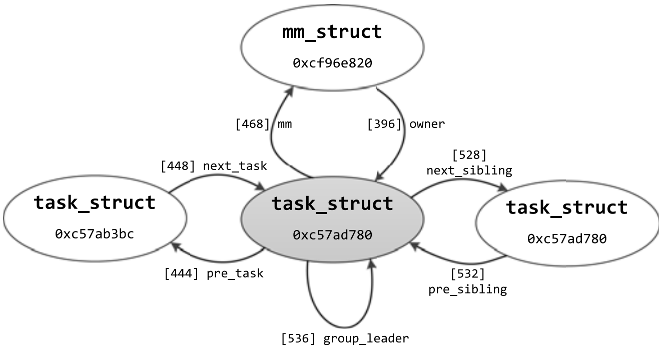

Fig. 4. Some loop path invariant examples for `task_struct` instances `0xc57ad780` in Linux kernel 2.6.26. Note that $[i]$ annotates the ith offset of the specific field in the corresponding data structure definition.

Loop invariants widely exist. When we follow the pointer path to traverse kernel object, we often find loop invariants widely exist. For example, as illustrated in Fig. 4, for a `task_struct` instance at address `0xc57ad780` from Linux kernel 2.6.26, we could have a loop path from `mm` field, to get an instance `0xcf96e820`, from which we follow its `owner` field and come back to the original `task_struct` instance at `0xc57ad780`. Meanwhile, we could have another loop by traversing the task list, that is the next task of its previous task is this task, and similar favor for children list and sibling list. Moreover, not only just `task_struct` has such a loop path, but also many other kernel data structures such as memory, file system, network communication, and device drivers all do contain such invariants.

Loop invariants are distinct across different kernels. To be used as signatures, the loop-invariant must be distinctive across different kernels. Our observation with a wide range of OS kernels confirms that these loop invariants are unique. Note that adding/removing one single offset in a data structure will lead to an entirely different loop.

Loop invariants are self-certifying. More importantly, we observe that a loop invariant is self-certifying. In particular, it has a mathematic foundation from the fixed point function $x = f(x)$ view. While testing whether x is true, we could test whether $f(x)$ is true by simply comparing the value of x and $f(x)$. Note f could be defined using pointer dereferencing $(*)$ and arithmetic operations $(+, -)$. For example, for a `task_struct` instance x when following the `mm_struct` path, we could have a $f(x) = *(*(x + 468) + 396) = x$. Theoretically, it is unlikely with a probability of $(1/2^{32})^2$ for a 32 bit machine that there are two memory chunks which happen to have such a loop.

Summary. Therefore, the essential goal of our data signature approach in OS-Sommelier$^+$ is to automatically identify these loop invariants, derive distinctive signatures, and make them faster for fingerprinting OS. Next, we present the detailed design of how to extract the loop invariant in the training phase, and how to scan the loop invariant in the matching phase.

4.2 Structural Invariant Extraction

As loop invariant represents a fact for a data structure and our goal is to extract them for kernel fingerprinting. To this end, we first need to extract the kernel data structure information (Section 4.2.1), from which to build the data structure graph (Section 4.2.2) and derive the loop invariants (Section 4.2.3). While we could use a compiler to extract the data structure definition from source code, we use a debugger assisted approach. In the following, we present our detailed design.

4.2.1 Data Structure Extraction

To extract the kernel data structure in the training phase, we assume we have debug information which is generated when compiling Linux kernel, and all the address mappings of global variables which comes from the `System.map` file. The data structure graph is rooted from kernel global variable. To achieve this, we leverage a debugger, in particular the `crash` [1] tool, to manage the debug symbol and return the data structure types, and we call this internal component *Type Querier*. We cannot directly use `crash` for a number of reasons: (1) `crash` does not expand the embedded `struct` when querying the data structure types, and does not expand the array as well; (2) `crash` does not tell how to type the `union` field and the destination of `void` pointers; and (3) `crash` only returns the syntactic definition of a field. Thus, we have to extend `crash` to achieve our goals. In particular, we extend `crash` with the following new capabilities:

1) *Expanding the embedded* `struct`. The goal of expanding the embedded structure is to get a global offset starting from 0 for the first field, for all the field within a structure. To expand the embedded structure, we check each field F_i, which is the ith-field returned by `crash` when querying the data structure definition. If F_i is an embedded structure, we then recursively check whether the next layer field in F_i is an embedded structure or not. If so, we update their offset for each field using a global offset. To identify a field is embedded, we check the size of that field: if it is greater than 4, we will normalize the type for that field, and further check if it is a user defined composite data type or not. The intuitions is that the primitive type except `double` are always within the size of 4 bytes (note our software running on a 32-bits ×86 architecture), and hence all others with size greater than 4 need to be further checked.
2) *Expanding the array element*. When querying `crash`, the returned array type is also encapsulated. Similar to the embedded `struct` expanding, we expand the array element as well, and we annotate each element with their detailed types and a global offset.
3) *Cloning the* `union` *type*—Since each field in a `union` type begins at the same location in memory, if F_i is a `union` type, we need to copy each field to the location of F_i. In other words, each field in `union` gets a slot at the same offset. Also, if a `union` contains user-defined structure, we will expand that structure as well.
4) *Identifying the pointer field*. To identify pointer field, we have to normalize F_i to its primitive form or one standard form for user-defined types. We cannot directly identify pointers just based on the syntactic form of a field because programmers could use `typedef` to define pointer field and type aliases,

and we need to normalize each type to the known primitive type or user-defined struct type (i.e., eliminate the type alias), which is stored in K. Then, we could just check the existence of $*$ in the normalized form to identify the pointer field.

5) *Resolving the specific type for `void` pointers.* When a memory holds a `void` pointer, we have to stop because we do not know the data structure type at the destination address. Fortunately, if the destination address is within the known data object, we can still type the `void` pointers. Therefore, we maintain a data object pool. Whenever we resolve a memory address with concrete types, we will insert the instance of this data structure to our object pool. If the `void` pointer points to the object which does not exist when we query the type, we will return its type unresolved temporarily, and push it to a work-list for future resolution (because the pool increases whenever we identify a new object). If in the end, we still cannot resolve its type, we will delete it from the work-list and return its type cannot be resolved.

As a result, every time when asking for a data structure definition, we will return a *flattened* data structure type definition, which has expanded all the embedded data structure and array element with global offset for each field, cloned all the union field, identified all the pointer field, and resolved the `void` pointer as much as it can at that moment, for a given memory address. These information will be used in our memory graph builder.

4.2.2 Memory Graph Builder

As illustrated in Section 4.1, to extract the loop invariant, we have to traverse a memory graph, which reflects a point-to relation between objects and their fields. The root of our memory graph is the data object address defined in `System.map` file, a symbol table file used by Linux kernel. While a symbol is a variable name or a function name of a program, and `System.map` contains the symbol address, symbol type (such as data or code symbol), and symbol name. After we get all the global addresses for each object from `System.map`, we keep them in a work list. Next, we iterate to pick up one and build a memory graph.

We also use an algorithm shown in Algorithm 3 to illustrate how to build the memory graph for Linux kernel data structures. When given a memory address a and its type t, our `Graph_Builder` first allocates space for G (line 2), keeps the address, type and size (lines 3-5), and then updates the object pool to contain the newly identified object (line 6) with the memory address and its type. Next it calls our *type querier* to return a data structure definition for a ($TQ(a, t)$ at line 11)), and then we iterate each field F_i returned in $TQ(a, t)$ (Type Querier described in Section 4.2.1) to store the type and address of each field in the allocated space $node_i$ (lines 8-10). Also, we check whether or not F_i is a pointer field (line 11). If so, we keep the point-to $value$ (i.e., the destination address) of this field (line 13). Note the d_{addr} is acquired by deference the memory address in the snapshot. For the graph edge, there are two cases: (1) if there is a graph containing d_{addr}, then we just connect the two nodes (line 15), (2) otherwise, we will first recursively call `Graph_Builder` to build a sub-graph G' for d_{addr} with type $T(F_i)$ (line 17), and then connect the $node_k$ in G' with $node_i$ (line 18). Why there exist two cases is because not all the edge is connected to the root of a graph, for example, the next pointer at offset 180 of `task_struct` shown in Fig. 4.

Algorithm 3. Memory Graph Builder

Require: $TQ(a, t)$ returns the type definition for destination address a, with the type t of a; $O(F_i)$ returns the offset of F_i in the checked data structure.
Input: the memory address a and its type t;
Output: G, which is a memory graph reachable from a;

```
1:  $G$ Build_Graph($a, t$)
2:  $G \leftarrow$ new $G$
3:  $G.type \leftarrow t$
4:  $G.addr \leftarrow a$
5:  $G.size \leftarrow \mathbf{sizeof}(\mathbf{a})$
6:  $ObjPool \leftarrow ObjPool \cup \{<a, t>\}$
7:  for each $F_i \in TQ(a, t)$ do
8:    $G.node_i \leftarrow$ new $node$
9:    $G.node_i.type \leftarrow T(F_i)$
10:   $G.node_i.addr \leftarrow a + O(F_i)$
11:   if $P(F_i)$ then
12:     $d_{addr} \leftarrow (*(a + O(F_i)))$
13:     $G.node_i.value \leftarrow d_{addr}$
14:     if $\exists G'$ $s.t.$ $d_{addr} \equiv G'.node_j.addr$ then
15:       $G.node_i.edge \leftarrow <G.node_i, G'.node_j>$
16:     else
17:       $G' \leftarrow$ Build_Graph($d_{addr}, T(F_i)$)
18:       $G.node_i.edge \leftarrow <G.node_i, G'.node_k>$,
          where $G'.node_k.addr \equiv d_{addr}$
19:     end if
20:   end if
21: end for
22: return $G$
```

As presented in Algorithm 3, we use a depth-first search (DFS) to build the memory graph (line 17). However, in practice, we cannot keep searching because of the scalability issues, as kernel usually has a large number of connected objects from a given object. Thus, we limit our search depth to a threshold of 4 when we building the memory graph for each kernel global variable exported in `System.map`. This threshold is chosen based on the recent result from SigGraph [26] that demonstrates we can use a two layer signature of a `task_struct` to infer a unique `task_struct` object. Once we have built a memory graph G for each data object, our next step is to extract the loop invariants from G. The details on how to extract them is presented next.

4.2.3 Loop Invariant Extractor

Since a loop invariant identifies a loop path in a memory graph, our *Loop Invariant Extractor* then performs a graph traversal to extract them. To make our system simple and scalable, we use a limited length of breadth-first search (BFS) algorithm to identify the loop. To be more specific, we outline our BFS based loop path detection in Algorithm 4. At a high level, given a graph node G for data structure instance, we incrementally check if a newly connected path whether or not can lead to a loop path to G; if so, we output such a loop path, otherwise, we keep checking until no connected node can be added or the path length exceeds the threshold.

Algorithm 4. Loop Invariant Extraction

```
Input: a memory graph $G$ (of object $a$ whose type is $t$);
Output: all loop path starting from and ending to $G$ with
        the path length limited to $L$;
 1: for each $node_i \in G.node$ do
 2:   $Conn\_node \leftarrow Conn\_node \cup \{node_i\}$
 3: end for
 4: $path\_length \leftarrow 0$
 5: while $path\_length < L$ do
 6:   for each $node_i \in Conn\_node$ do
 7:     if $\exists edge_k = <node_i, G'.node_j>$ then
 8:       if $G'.node_j \notin Conn\_node$ then
 9:         $Conn\_node \leftarrow Conn\_node \cup \{G'.node_j\}$
10:       else
11:         if $G' \equiv G$ then
12:           output the loop path from
              $<G.node_m, G''.node_u>$ to
              $<G'''.node_v, G.node_j>$
              where $G''.node_u \in Conn\_node \wedge$
              $G'''.node_v \in Conn\_node$
13:         end if
14:       end if
15:     end if
16:   end for
17:   if no $node_i$ added to $Conn\_node$ then
20:     return
19:   end if
20:     $path\_length$ ++
21: end while
```

As illustrated in Algorithm 4, we first use a set *Conn_node* to keep all the initial node of G (lines 1-3), then we examine the newly checked edge (line 7), if the connected to node is a new node (line 8), we add it to *Conn_node* (line 7); otherwise, we then check if the graph G' this node belongs to is G or not (line 11), if so, we output the loop path, which is starting from and end to G (line 12). Note, every time we aim to detect the loop which is ended with G; for those other loop path, we will detect them when we check that particular graph node, and all the graph node G will be visited to detect the loop path for them. Also, all the data object will have a corresponding G created by using our Algorithm 3.

To detect all the loop path of a data structure instance, rooted from a graph G, suppose each graph node has N number of pointer fields, then the worst case complexity of our loop detection will be $O(N + N^2 + N^3 + \cdots + N^L) = O(N^L)$, where L is the threshold of the maximum path length. Suppose we have M number of objects, then the total complexity to detect all loop for all object will be $O(M \times N^L)$. Note in practice, N is usually less than 5, and also our system is used in an offline fashion to extract the loop invariant and it is a one time execution.

4.2.4 Adding More Signatures

As discussed in Section 2.2.2, we also extract the *size invariant* of a data structure as one of the signatures. For instance, the `task_struct` of Linux kernel usually has over hundreds of fields, up to thousands of bytes. If we can precisely identify the size of this data structure in memory, it could be served as a strong signature.

However, the challenging lies in how to identify the size of a data structure (e.g., `task_struct`). Fortunately, we have an observation that the same type of kernel data structure tends to be allocated in a pool, the size of a data structure can be determined based on the minimum distance between two identified data structures of the same type. For instance, suppose we have identified all instances of `task_struct`, we can then compute the shortest distance between these instances. The minimum value would be the upper bound of the `task_struct`. Then, we need to have an approach to identify the instance of `task_struct`. Fortunately, it has many *loop invariants*. Therefore, our design is to first scan the instances of `task_struct` by using the *loop invariant*, and then compute the size.

4.3 Structural Invariant Scanning and Matching

We use a brute force scanning of the *structural invariant* (in particular *loop invariant*) to identify the data structure instances, and derive other signatures (such as the *size invariant*). The scanning is pretty straightforward: starting from the beginning of kernel address space, we check whether a given address contains a *loop invariant* by following the pointer dereference and the offset information encoded in the signatures. If it matches, we output this instance. After we have scanned all the instances, we then derive *size invariant*. By combining all the invariants, we can pin-point the OS versions.

Note that the loop invariant extraction is done during a training phase. After we extracted all the invariants for all the kernels, we will organize them in a decision tree. When we scan the guest OS memory, we can then quickly tell which signature the kernel matches.

5 Evaluation

We have implemented OS-Sommelier$^+$, with 7.5 K lines of code. In particular, we implemented the code signature approach with 4.5 K lines of C code and its *correlative disassembler* is built on top of XED [3] library, and we implemented the data signature approach with 3 K lines of python code, atop a kernel dump analysis tool crash [1]. In this section, we present our experimental result. We first tested its effectiveness in Section 5.1, with 27 Linux kernels from 2.6.26 to 3.12.0, and then we report the performance of OS-Sommelier$^+$ in Section 5.2

5.1 Effectiveness

To test OS-Sommelier$^+$, we have to execute it in two phases and have the following experimental setup:

- *Ground-truth collection phase*. To generate the ground-truth signature for each testing OS kernel, we used one physical memory dump for both code

TABLE 2
Testing Memory Snapshot Configuration

Configuration	VMware	QEMU	KVM	Xen	VirtualBox
Mem Size (MB)	256	512	512	768	1024
Time (min)	5	10	10	15	20

Time is calculated right after the VM is power on.

signature and data signature approach. Normally the ground truth is collected by a cloud provider or a forensics examiner with a one-time effort. To obtain the physical memory dumps, we run each of the OSes in a QEMU [2] VM with 512M bytes RAM (131,072 pages with 4 K bytes each). After the guest OS booted up, we took a memory dump to compute the ground truth.

- *Testing phase*. For the deployment testing, we collected five memory dumps for each testing OS with different VMs (including VMware Workstation, KVM, Xen and VirtualBox) at different moments after OS booted. As shown in Table 2, we also varied the VM configurations with different physical memory size (from 256 M to 1 G) and took the snapshot at different time after the VM is power on. The page swapping is enabled for these memory dumps in order to evaluate the robustness of our signature scheme in real scenario. Our host machine has an Intel Core i7 CPU with 8 G memory machine, installing a Ubuntu 11.04 with Linux kernel 2.6.38-8.

In the following, we present the detailed experimental results for these two phases.

5.1.1 Ground-Truth Collection

Table 3 presents the detailed experimental results for each tested OS in our ground truth collection. We start the Linux kernel from version 2.6.26, and end at the version 3.12.0. These are the kernels released in the past five years. For each kernel, we collect its code signature, and data signature. These signatures will be used in our testing phase (Section 5.1.2). More specifically.

Code signature approach. To generate the code signatures, there are two key components involved, namely *Kernel Code Identification* and *Code Signature Generation*. We report the step-by-step output by these two components below:

- *Kernel code identification*. There are three steps involved in kernel code identification (Section 3.1). We report the output of these steps respectively. As presented in column $|C_K|$, we usually could identify 28.93 clusters based on whether any two contiguous kernel pages share identical PTE and PDE bits, and by searching whether the cluster contains the 6-bytes TLB flushing sequence, we can further narrow down the core kernel code to at most two clusters (column

TABLE 3
Experimental Result of OS-SOMMELIER$^+$ during the Ground Truth Signature Generation

OS-kernels	Code Signature Approach										Data Signature Approach		
	$\|C_K\|$	$\|C_{Kk}\|$	#Pages	$\|T\|$	#Pages'	#DisPage	$P_r\%$	$D_r\%$	$\|S\|$	$\|S'\|$	#Loop Invariant	$\|task\_struct\|$	#Reached Struct
Linux-2.6.26	69	1	812	2	811	526	64.86	9.48	526	1	179	3180	40
Linux-2.6.27	72	1	845	2	844	548	64.93	9.57	548	1	75	3180	42
Linux-2.6.28	109	1	885	2	884	575	65.05	9.78	575	1	72	3188	40
Linux-2.6.29	106	1	908	2	907	597	65.82	9.76	597	1	75	3132	37
Linux-2.6.30	119	1	1446	2	1445	918	63.53	11.1	918	1	75	3184	37
Linux-2.6.31	26	1	1545	3	1098	976	88.89	10.99	976	2	89	3220	35
Linux-2.6.32	26	1	1589	2	1588	1005	63.29	11.33	1005	2	89	3236	33
Linux-2.6.33	28	1	1606	3	1152	1039	90.19	11.23	1039	2	89	3248	33
Linux-2.6.34	28	1	1617	2	1616	1043	64.54	11.19	1043	2	88	3280	41
Linux-2.6.35	28	1	1640	3	1175	1056	89.87	11.35	1056	1	139	3244	41
Linux-2.6.36	28	1	1641	3	1183	1071	90.53	11.15	1071	2	88	3244	41
Linux-2.6.37	3	2	768	1	767	700	91.26	11.49	700	2	242	3228	42
Linux-2.6.38	3	2	768	1	767	697	90.87	11.51	697	2	133	3228	41
Linux-2.6.39	3	2	768	1	767	687	89.57	11.72	687	2	88	3228	41
Linux-3.0.0	3	2	768	1	767	689	89.83	11.87	689	1	88	3236	41
Linux-3.1.0	3	2	768	1	767	682	88.92	11.73	682	2	88	3232	43
Linux-3.2.0	3	2	768	1	767	680	88.66	11.6	680	2	88	3224	53
Linux-3.3.0	3	2	768	1	767	680	88.66	11.83	680	2	88	3236	55
Linux-3.4.0	3	2	768	1	767	679	88.53	11.74	679	2	88	3228	54
Linux-3.5.0	5	2	768	1	767	676	88.14	11.87	676	2	90	3232	53
Linux-3.6.0	5	2	768	1	767	690	89.96	11.82	690	1	88	3216	53
Linux-3.7.0	5	2	768	1	767	679	88.53	11.74	679	1	88	3224	56
Linux-3.8.0	5	2	768	1	767	674	87.87	11.53	674	1	159	3252	42
Linux-3.9.0	7	2	768	1	767	660	86.05	11.9	660	1	160	3256	43
Linux-3.10.0	8	2	768	1	768	687	89.45	11.72	687	1	157	3260	43
Linux-3.11.0	8	2	768	1	768	688	89.58	11.81	688	1	155	3256	43
Linux-3.12.0	75	1	1942	1	1942	1243	64.01	11.37	1243	1	133	3268	42
Mean	28.93	1.56	1036.89	1.56	968.60	772.04	81.91	11.27	772.04	1.49	110.78	3227.41	43.15

$|C_{Kk}|$). The largest cluster in C_{Kk} contains on average 1,641 kernel code pages, and this number is reported in the fourth column.

- *Code signature generation*. We report seven categories (from the fifth to 11th column) of data for this component. In particular, in the fifth column, we report the total number of clusters $|T|$ identified by connecting the pages with direct forward function call constraint; the sixth column (#pages') reports the total number of pages in the largest cluster of $|T|$. For these pages, how many of them (i.e., #DisPage) can be disassembled is reported in the seventh column, and this ratio P_r reported in the eighth reflects the effectiveness of our correlative disassembling. For the disassembled pages, how many bytes in each page can be disassembled (i.e., the disassembled byte ratio D_r) is reported in the 9th column and this ratio estimates how much information we eventually retained after all of our transformations. Finally, we report the total number of signatures generated in column $|S|$ (the 10th column). Among all of these signatures, at least how many of the signatures $|S'|$ (reported in 11th column) we need to check in order to determine the corresponding kernel version.

  From the table, we could see that on average we will further divide the core kernel code into 1.56 clusters. The largest cluster contains on average 968.60 pages. Among these pages, on average 772.04 of them can be disassembled with a ratio (P_r) of 81.91 percent. For each disassembled page, we covered 11.27 percent of bytes during the disassembling. We have to emphasize that our goal is not for accurate disassembly, and the disassembling rate is not a crucial factor for the quality of our OS fingerprinting. In comparison, the disassembled page ratio (P_r) make more sense, because it shows how many different pages have contributed to the signatures. The final signature cluster size on average is 772.04 MD5-hashes. Meanwhile, for all the hashes we generated, the majority of them only needs one of the page hash to uniquely identify the target OS.

Data signature approach. In ground truth generation, OS-SOMMELIER$^+$only involves the *Structural Invariant Extraction*component. We also reported the statistics of the extracted structural invariant in the last three columns of Table 3. As discussed in Section 4.1, *loop invariant* is so strong, and it can be used to fingerprint OS kernel, when combined with other strong signatures such as *size invariant* (Section 4.2.4).

Nearly for all the running Linux kernels, it must contain the process control structure, namely, `task_struct`. Therefore, we extracted all the possible *loop invariants* starting from `task_struct`. This number is reported in the 12th column of Table 3. We can see on average, we could have 110.78 number of the loop invariants for `task_struct`. For the size of the `task_struct`, it is reported in the 13th column. We can notice that `task_struct` often has a very large size, with an on average of 3,227.41 bytes. We also report how many other data structures can be reached by the loop invariants, and they are presented in the last column. We can notice that the loop invariants of `task_struct` can often reach 43.15 other data structures.

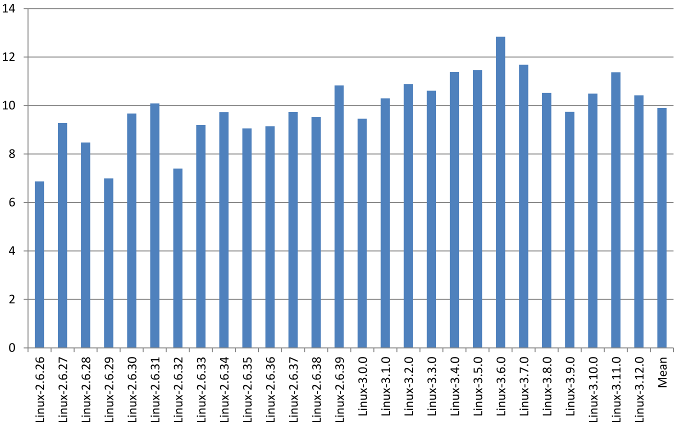

Fig. 5. Scanning time of OS-SOMMELIER$^+$ w/ code sig.

Again, we would like to emphasize that there could be other loop invariants from other data structures. Because they are usually so strong, in this paper, we only derived the ones from `task_struct`. When combined with the strong size invariants, they can be used to precisely fingerprint all of these kernels we used for the derivation.

5.1.2 Testing Phase

In our testing phase, we used two sets of kernels to evaluate OS-SOMMELIER$^+$. The first set contains the exact 27 kernel in our ground-truth generation, and the second set contains the same kernel as in the first set but recompiled with stack frame omission option (this is to test the code agnostic feature of our data signature). As a result, in total we have 54 kernels. We execute these kernels in a diversified VMs using different configurations, as reported in Table 2.

We then took a snapshot for each kernel, and sent to OS-SOMMELIER$^+$ for the fingerprinting. Without any surprise, our code signature approach successfully identified the 27 known kernels in the training set. However, it cannot identify these recompiled one. Fortunately, out data signature identified all the kernels without any false positive and false negatives.

5.2 Performance Evaluation

Next, we measured how fast to execute OS-SOMMELIER$^+$ to fingerprint a kernel. As illustrated in Fig. 1, OS-SOMMELIER$^+$ will first invoke code signature approach to determine the kernel version (because of its fast speed). Only when it fails, it then invokes the data signature approach.

Code signature approach. To match a given kernel, code signature approach first needs to identify the core kernel code, perform the robust assembly, hash the distilled code, and then perform the matching. The execution time for all the identified 27 kernels is reported in Fig. 5. We can notice that on average it took 9.9 seconds to fingerprint a known kernel.

Note that while our OS-SOMMELIER$^+$ uses KMP string matching algorithm [24] to match the final signature string with the ground truth signature string (which has hundreds of page hash values), we can in fact use only few page hashes to uniquely pinpoint the exact OS. We confirmed this observation and verified that in most cases, we can just use one MD5 to differentiate the kernel (as shown in the $|S'|$

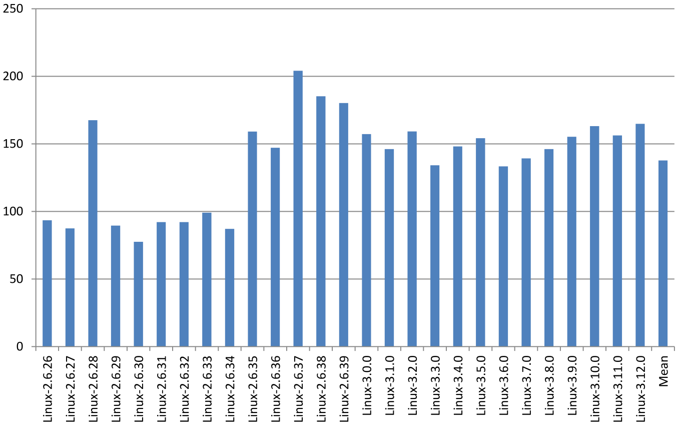

Fig. 6. Scanning time of OS-Sommelier$^+$ w/ data sig.

column in Table 3), thanks to our strong, sensitive code hash based signatures.

Data signature approach. When our code signature cannot tell the kernel version, OS-Sommelier$^+$ then invokes the data signature approach. This component took much longer time than that of code signature. This is because it uses a brute force memory scanning to test each potential memory address whether or not matching with an existing known loop invariant. Also, it has to test with all the known loop invariants in the signature data base. Therefore, it took on average 137.8 seconds for the 27 unknown kernels which is reported in Fig. 6, and this overhead also includes the time spent by the code signature approach because it only gets involved when code signature approach fails.

6 Discussion

Given the threat model described in Section 2.1, in this section, we discuss the possibility of various evasion attacks and the limitations of OS-Sommelier$^+$.

Creating extra noisy data. As discussed in our threat model, we assume that the integrity of main kernel code in the VM can be enforced by the cloud provider. After compromising a VM, the adversary cannot directly modify the main kernel code to evade our OS fingerprinting. She can only generate extra fake data to mislead our system. For example, by exploiting our kernel code identification process, the adversary may make up some data to make OS-Sommelier$^+$ believe that some of the fake pages are kernel code.

To counter this kind of attack, we can configure OS-Sommelier$^+$ to examine all possible kernel code clusters. As the true kernel code remains intact, it will still be correctly identified to be one of the clusters. Then instead of picking the biggest cluster for signature matching, we can check all the clusters one by one and report the recognized OS version respectively.

Obfuscating the kernel code or data structure. If a VM user is uncooperative or malicious, she can obfuscate the kernel code or data structure to bypass our fingerprinting. To name a few, she can mess with function prologues to disrupt our correlative disassembling if the adversary can access the kernel source code. She can also manipulate direct function calls and indirect function calls to confuse our kernel code identification process. She can even applies for kernel data structure randomization if she has the source code.

In general, this kind of evasion attacks is equivalent to fingerprinting a completely new and unknown OS, which we consider to be out of our current scope. Meanwhile, cloud providers need to maintain an up-to-date signature database for all the kernels including with new patches. Otherwise, OS-Sommelier$^+$ may not be able to recognize it.

7 Related Work

Network protocol based fingerprinting. OS fingerprinting has initially been investigated from a network protocol perspective. Inspired by earlier efforts leveraging TCP stacks to find (1) protocol violations, (2) vendor-specific design decisions [11], and (3) TCP implementation differences [28], TCP based fingerprinting was proposed. The basic approach is to actively send carefully created TCP/IP or ICMP packets to the target machine, analyze the differences in the response packets, and derive fingerprints and match the database with known OS fingerprints. Nmap [16], Xprobe2 [4]/Xprobe2++ [40], and synscan [36], are all such tools based on network based fingerprinting. Besides the active packet probing fingerprinting, there are other passive OS fingerprinting techniques through sniffing network packets such as p0f [34].

The rationale behind network-based fingerprinting is that different OSes tend to have different implementations for certain network protocols and services. While this rationale is often true, it is not accurate enough to distinguish the minor versions of an OS kernel, because minor OS versions may have the same protocol implementations. Moreover, the network-based fingerprinting may not be applicable in many new application scenarios, such as memory forensics.

CPU-based fingerprinting. Recently Quynh proposed a system called UFO [30] to fingerprint the OS of a virtual machine in the cloud computing environment. It explores the discrepancies in the CPU state for different OSes. The intuition is that in protected mode, many CPU registers such as GDT, IDT, CS, CR, and TR, often have unique values with respect to different OSes. Thus, by profiling, extracting, and differing these values using a VM monitor (VMM), UFO generates unique signatures for each OS.

While UFO is effective and efficient in fingerprinting certain OS families (e.g., Linux kernels, it does not work well for many other OS families, such as the Windows OS and close versions of Linux kernels. Moreover, the requirement of access to the CPU state cannot be always met for various application scenarios (such as memory forensics and kernel dump analysis).

Filesystem-based fingerprinting. In a cloud VM management scenario, it would be straightforward to identify the OS version by examining the file system of the virtual machine. For example, one can look for the kernel code on the file system and check its hash. In fact, tools such as virt-inspector [23] already support this capability.

However, this approach is not feasible for a machine with encrypted file systems. Also, the requirement of access to the file system may not be viable for some applications such as memory forensics when only having a physical memory dump.

Memory-based fingerprinting. Recently a couple of techniques were proposed to utilize the memory data for OS fingerprinting, but none of them can simultaneously achieve all of the three design goals: efficiency, accuracy, and robustness. In particular, Christodorescu et al. [10] proposed to use the interrupt handler for OS fingerprinting, because the interrupt handler varies significantly across different OSes. While efficient, this approach is not accurate enough to differentiate Windows XP kernels with different service packs, and it also cannot pinpoint some of the FreeBSD and OpenBSD kernels [19]. Meanwhile, this approach also requires access to CPU registers because they directly identify the interrupt handler from the IDT register. Again, it is not directly suitable for applications such as memory forensics when only memory is available.

Most recently, SigGraph [26] was proposed as a graph signature scheme to reliably identify kernel data structures from a memory dump. As different OSes tend to have different data structure definitions, it has been demonstrated that SigGraph has the potential for OS fingerprinting [26]. However, SigGraph is far from being efficient, as it scans every pointer and non-pointer field and examines the data structures in many hierarchical levels. In contrast, OS-SOMMELIER$^+$ employs *loop invariant* which significant reduces the verification time when scanning data structure instances. Also, OS-SOMMELIER$^+$ supports the derivation of *size invariant* which is another strong signatures.

8 CONCLUSION

We have presented the design, implementation, and evaluation of OS-SOMMELIER$^+$, a multi-aspect, memory exclusive, and robust approach for precise and efficient OS fingerprinting in the cloud. The key idea is to first compute the core kernel code signatures to precisely fingerprint an OS, and the precision and efficiency are achieved by our core kernel code identification, correlative disassembling, code and signature normalization, and resilient signature matching techniques. If code hash fails, it then uses a strong loop invariant and size invariant to fingerprint the OS from the data perspective. Our experimental result with 27 Linux kernels shows that the code signature of OS-SOMMELIER$^+$ can precisely fingerprint all known OSes in a fast fashion, but it is too sensitive to kernel code changes, and the data signature of OS-SOMMELIER$^+$ complements the code signature and can fingerprint even unknown kernels (e.g., the kernel with different code but the same data structure).

REFERENCES

[1] Crash. (2014) [Online]. Available: http://mclx.com/projects/crash/

[2] QEMU: An open source processor emulator. (2014) [Online]. Available: http://www.qemu.org/

[3] Xed: X86 encoder decoder. (2014) [Online]. Available: http://www. pintool.org/docs/24110/Xed/html/

[4] O. Arkin, F. Yarochkin, and M. Kydyraliev, "The present and future of xprobe2: The next generation of active operating system fingerprinting. sys-security group," Jul. 2003.

[5] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 38–49.

[6] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Security Symp.*, 2003, pp.105–120.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th Conf. USENIX Security Symp.*, 2005, p. 17.

[8] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. 8th Workshop Hot Topics Operating Syst.*, 2001, pp. 133–138.

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 2–13.

[10] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni, "Cloud security is not (just) virtualization security: A short paper," in *Proc. ACM Workshop Cloud Comput. Security*, 2009, pp. 97–102.

[11] D. E. Comer and J. C. Lin, "Probing tcp implementations," in *Proc. USENIX Summer 1994 Tech. Conf.*, Boston, MA, USA, 1994, pp. 245–255.

[12] G. Conti, S. Bratus, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg, R. Perez, and A. Shubina, "Automated mapping of large binary objects using primitive fragment type classification," in *Proc. DFRWS Annu. Conf.*, 2010, pp. 3–12.

[13] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection," in *Proc. IEEE Symp. Security Privacy*, San Francisco, CA, USA, May 2012, pp. 586–600.

[14] Y. Fu and Z. Lin, "Bridging the semantic gap in virtual machine introspection via online kernel data redirection," *ACM Trans. Inf. Syst. Security*, vol. 16, no. 2, pp. 7:1–7:29, 2013.

[15] Y. Fu and Z. Lin, "Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery," in *Proc. 9th Annu. Int. Conf. Virtual Execution Environ.*, Houston, TX, USA, Mar. 2013, pp. 97–110.

[16] Fyodor. (2007, Jan.). Remote os detection via TCP/IP fingerprinting (2nd generation). insecure.org [online]. Available: http://insecure.org/nmap/osdetect/

[17] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2003.

[18] L. G. Greenwald and T. J. Thomas, "Toward undetected operating system fingerprinting," in *Proc. 1st USENIX Workshop Offensive Technol.*, 2007, pp. 1–10.

[19] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin, "Os-sommelier: Memory-only operating system fingerprinting in the cloud," in *Proc. 3rd ACM Symp. Cloud Comput.*, San Jose, CA, USA, Oct. 2012, pp. 5:1–5:13.

[20] S. Hand, Z. Lin, G. Gu, and B. Thuraisingham, "Bin-carver: Automatic recovery of binary executable files," in *Proc. 12th Annu. Digit. Forensics Res. Conf.*, Washington, DC, USA, Aug. 2012, pp. 108–117.

[21] Intel-64 and IA-32 architectures software developer's manual combined volumes 3A, 3B, and 3C: System programming guide, parts 1 and 2: 11-28, 2012.

[22] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proc. 14th ACM Conf. Comput. Commun. Security*, Alexandria, VA, USA, Nov. 2007, pp. 128–138.

[23] R. W. Jones and M. Booth, Virt-inspector—Display operating system version and other information about a virtual machine. (2012) [Online]. Available: http://libguestfs.org/virt-inspector.1.html

[24] D. E. Knuth, J. H. Morris Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[25] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. 13th Conf. USENIX Security Symp.*, San Diego, CA, USA, 2004, pp. 255–270.

[26] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," presented at the *Proc. 18th Annual Network and Distributed System Security Symp.*, San Diego, CA, USA, Feb. 2011.

[27] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 143–158.

[28] V. Paxson, "Automated packet trace analysis of TCP implementations," in *Proc. ACM SIGCOMM*, 1997, pp. 167–179.

[29] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proc. 23rd Annu. Comput. Security Appl. Conf.*, Dec. 2007, pp. 385–397.

[30] N. A. Quynh, "Operating system fingerprinting for virtual machines," in *Proc. DEFCON 18*, 2010.

[31] A. Saberi, Y. Fu, and Z. Lin, "Hybrid-bridge: Efficiently bridging the semantic-gap in virtual machine introspection via decoupled execution and training memoization," presented at the 21st Annu. Network and distributed system security symposium, San Diego, CA, USA, Feb. 2014.

[32] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proc. 9th Working Conf. Reverse Eng.*, 2002, pp. 45–54.

[33] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proc. 21st ACM SIGOPS Symp. Oper. Syst. Principles*, 2007, pp. 335–350.

[34] C. Smith and P. Grundl, "Know your enemy: Passive fingerprinting. Identifying remote hosts without them knowing," Tech. Rep., Honeynet Project, Ann Arbor, MI, USA, 2002.

[35] A. Sotirov, "Hotpatching and the rise of third-party patches," in *Proc. Black Hat Tech. Security Conf.*, Las Vegas, NV, USA, Aug. 2006.

[36] G. Taleck, "Synscan: Towards complete tcp/ip fingerprinting," presented at the Canada Security West Conf., Vancouver, BC, Canada, 2004.

[37] P. Team, Pax address space layout randomization (ASLR). (2003) [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[38] A. Walters, The volatility framework: Volatile memory artifact extraction utility framework. (2014) [Online]. Available: https://www.volatilesystems.com/default/volatility

[39] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *Proc. 22nd Int. Symp. Reliable Distrib. Syst.*, 2003, pp. 260–269.

[40] F. Yarochkin, O. Arkin, M. Kydyraliev, S.-Y. Dai, Y. Huang, and S.-Y. Kuo, "Xprobe2++: Low volume remote network information gathering tool," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2009, pp. 205–210.

[41] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 203–216.

Yufei Gu is a third year PhD student in computer sciences at the University of Texas at Dallas. His research interests include memory forensics, program analysis, cloud computing, and virtual machine introspection.

Yangchun Fu is a fourth year PhD student with computer sciences at the University of Texas at Dallas. His research interests include systems security with a focus on program analysis and reverse engineering techniques, and their applications to virtual machine introspection (VMI) and cloud management.

Aravind Prakash graduated with MS degree in computer science from the University of Miami, FL, in 2009 and is working toward the PhD degree in system security at Syracuse University, NY. His research interests include program analysis, memory forensics, exploit diagnosis, and mobile security.

Zhiqiang Lin received the PhD degree from Purdue University in 2011. He is an assistant professor with the Computer Science Department, University of Texas at Dallas. His current research focuses on system and software security with an emphasis on binary code reverse engineering, vulnerability discovery, malicious code analysis, and OS kernel protection. He is a member of the IEEE.

Heng Yin received the PhD degree in computer science from the College of William and Mary in 2009. He is an assistant professor at the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, New York. His research interests lie in computer and network security. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.