OS-SOMMELIER: Memory-Only Operating System Fingerprinting in the Cloud

Yufei Gu[†], Yangchun Fu[†], Aravind Prakash[‡], Zhiqiang Lin[†], Heng Yin[‡]

[†]Department of Computer Science
The University of Texas at Dallas
800 W. Campbell RD
Richardson, TX 75080
{firstname.lastname}@utdallas.edu

[‡]Department of Computer Science
Syracuse University
400 Ostrom Avenue
Syracuse, NY 13210
{arprakas,heyin}@syr.edu

ABSTRACT

Precise fingerprinting of an operating system (OS) is critical to many security and virtual machine (VM) management applications in the cloud, such as VM introspection, penetration testing, guest OS administration (e.g., kernel update), 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 is always present in all these applications, in this paper, we present OS-SOMMELIER, a memory-only approach for precise and efficient cloud guest OS fingerprinting. Given a physical memory dump of a guest OS, the key idea of OS-SOMMELIER is to compute the kernel code hash for the precise fingerprinting. To achieve this goal, we face two major challenges: (1) how to differentiate the main kernel code from the rest of code and data in the physical memory, and (2) how to normalize the kernel code to deal with practical issues such as address space layout randomization. We have designed and implemented a prototype system to address these challenges. Our experimental results with over 45 OS kernels, including Linux, Windows, FreeBSD, OpenBSD and NetBSD, show that our OS-SOMMELIER can precisely fingerprint all the tested OSes without any false positives or false negatives, and do so within only 2 seconds on average.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.7 [**Operating Systems**]: Organization and Design; D.4.m [**Operating Systems**]: Miscellaneous

Keywords

Operating System Fingerprinting, Virtual Machine Introspection, Memory Forensics, Cloud Computing

SOCC'12, October 14-17, 2012, San Jose, CA USA

1. INTRODUCTION

In an Infrastructure as a Service (IaaS) cloud, there is an increasing need for guest operating system (OS) fingerprinting. For each virtual machine (VM) running in the cloud, the cloud provider needs to know the exact OS version, such that OS-specific management and security tasks can be performed. For example, the cloud provider may need to perform virtual machine introspection (VMI [8, 14, 19, 27, 28]) to monitor the activities within the guest VM, in order to detect and prevent malicious attacks. The cloud provider may also conduct penetration testing to pinpoint the security vulnerabilities in the guest OS, and perform memory forensic analysis or kernel dump analysis on the memory snapshots of the virtual machine. All these tasks require prior knowledge of the *exact* guest OS version (especially for the recent binary code reuse based VMI techniques such as VMST [14]). Although the cloud users may provide the information about the OS version, such information may not be reliable and may become out dated after the guest OS is patched or updated on the regular basis.

Unfortunately, the existing OS fingerprinting techniques fall short in *precision* and *usability*. More specifically, network-based fingerprinting (e.g., nmap [15, 16] and Xprobe2 [3]) recognizes the discrepancies in network protocol implementations by sending crafted packets and analyzing the difference in responses. This network-based approach is often imprecise and cannot pinpoint the minor OS differences such as Linux-2.6.18 vs Linux-2.6.20. Moreover, the network-based approach becomes less usable since modern OSes (e.g., Windows 7) disable many network services by default (by closing the TCP/UDP ports).

Two other recent systems explore the end-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 families. Filesystem-based fingerprinting is another approach. Tools like virt-inspector [20] 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, due to its privacy concerns.

Therefore, to provide a strong support for management and security 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-only approach for OS fingerprinting. This new approach needs to be *precise* enough to recognize the minor versions of an OS kernel, and *efficient* enough to get the fingerprinting result within a short period of time. To

make this technique as generic as possible, we choose not to rely on other inputs (such as CPU state). As a result, our memory-only fingerprinting can provide direct support to many other applications such as memory forensics and kernel dump analysis.

To answer these needs, we have developed a new memory-only OS fingerprinting system, called OS-SOMMELIER. The key idea of OS-SOMMELIER is to compute unique *core* kernel code hashes as the signature from a memory dump to precisely fingerprint an OS. However, to realize this idea, we are facing several new challenges, especially for the widely used x86 architecture in the cloud. The first challenge is how to distinguish the main kernel code from the rest of code and data. It is commonplace that code and data can be mixed together. The device drivers should also be excluded from the signature, as they are not part of the main kernel code and can be loaded in different versions of the OS kernel. The second challenge is how to tolerate real-world issues such as address space layout randomization (ASLR [5, 6, 39, 41]), page swapping, and the dynamic kernel code patches (i.e., hot-patches [36]).

We have devised a suite of new techniques to address these challenges such as *core kernel code identification*, *correlative disassembling*, and *normalized signature matching* in our prototype system OS-SOMMELIER. Our experimental results with over 45 OS kernels, including Linux, Windows, OpenBSD, NetBSD, and FreeBSD (BSD family), show that our system can precisely fingerprint all the OSes we tested without any false positives and false negatives, in under 2 seconds on average.

The main contribution of this paper is highlighted as follows:

- We present a novel approach to precisely fingerprint an OS kernel when provided with *only* a physical memory dump. Our approach is general (OS-agnostic) 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, 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 have implemented our system OS-SOMMELIER, and tested it over a variety of widely used OS kernels such as Linux, Windows, and BSD family. Our experimental results show that OS-SOMMELIER has no false positives or false negatives for all the memory dumps in our data set. OS-SOMMELIER is also efficient. It takes just a few seconds to fingerprint a given OS.

2. SYSTEM OVERVIEW

In this section, we first describe the problem statement in §2.1, then identify the challenges and outline our corresponding key techniques in §2.2. Finally, we give an overview of our system in §2.3.

2.1 Problem Statement

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

- **Precision:** We need to determine the OS family and the exact version. For example, for Windows, we need to know whether it is Windows XP or Windows 7, and further identify which service pack has been installed. For Linux, we not only need to know its major version (e.g., 2.6 or 3.0) but also the minor version number. This is because many security tasks (e.g., [14, 19, 27, 28, 37]) have to rely on the exact OS version to make OS-specific decision.
- **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 just few seconds 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 bypass or mislead our fingerprinting system (to further defeat VMI for instance). Thus, our OS fingerprinting technique needs to be robust enough to counter various attacks. For example, an intuitive solution might be searching the OS version string or some particular byte sequence in the memory. Then attackers can easily tamper with this string to defeat the OS fingerprinting.

Threat Model and Our Assumption To achieve the goal of robustness, we need to consider a realistic thread model. As kernel rootkits become commonplace, we imagine that attackers can often obtain the highest privilege of the VM, infiltrate into the guest OS kernel and can execute arbitrary code and modify arbitrary memory locations within the VM. In order to defeat memory-based OS fingerprinting, attackers can manipulate the memory state (including the kernel data), by either modifying the existing kernel code and data, or creating extraneous noisy data. However, we assume the integrity of (at least the large body) the main kernel code. Recent advance in trusted computing techniques (e.g., [24, 32]) and virtual machine security (e.g., [4, 9, 43]) can easily ensure the integrity of the kernel code pages. For this reason, we decide to focus on the main kernel code to achieve robustness in memory-based OS fingerprinting.

2.2 Challenges and Key Techniques

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:

(1) A robust and generic way to identify the kernel page tables. Given a physical memory dump, the first step is to recover its virtual memory view. As virtual-to-physical address translation is conducted by looking up the page global directory (PGD) and page tables, it is necessary that we correctly identify PGDs from the physical memory dump. This step is required not only for this problem but also for many other applications such as code-reuse based semantic-gap bridging [14] and memory forensics [40].

To the best of our knowledge, none of the existing techniques including those in memory forensics (e.g., Volatility [40]) have provided a general solution for searching the PGDs. The current practices are very OS-specific. More specifically, it is either through profiling to get one exact PGD address, or by searching for the PGD field in process descriptors (by using process descriptor's signature) [40], or from the kernel symbols (e.g., retrieving the virtual address of `swapper_pg_dir` from the `system.map` file, and then subtracting with `0xc0000000` [19]).

Since our goal is OS fingerprinting, such OS-specific knowledge is not acceptable. Hence, we need a more generic way (i.e., OS-agnostic) to identify PGDs. To this end, we have identified the

inherent characteristics including the point-to relations in page tables, which have to hold true independent of specific OS versions. Based on these characteristics, we create a *generic PGD signature*.

(2) Correctly disassembling the kernel code. To compute hash values of the kernel code, we have to correctly disassemble it. However, it is widely known that correct code disassembly is still an open problem for x86 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 x86 instructions have varied lengths, a completely different instruction sequence will be disassembled if starting from a wrong instruction boundary.

To this end, we propose a *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 limitation, as there is a large amount of kernel code and the correctly identified portion is sufficient enough to serve the purpose of our fingerprinting.

(3) 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 for the device drivers and other loadable kernel modules. We aim to compute the hashes for the main kernel code only, because the presence of the other kernel modules is 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.

(4) Dealing with kernel address space layout randomization. To prevent kernel exploits, modern OSes (such as Windows Vista and Windows 7) have enabled address space layout randomization (ASLR [5, 6, 39, 41]) in the kernel space. Consequently, on each run, the base address of kernel code and data regions are randomized. This security feature poses a challenge to our signature generation and checking, because certain portions of the kernel code have to be updated in order to be relocated to a different place. More specifically, the data and code labels with their absolute addresses have to be updated.

To address this challenge, we introduce a *code normalization* process as well as a *signature normalization* process. Our *code normalization* will zero out these code and data labels from the disassembly. As such, the side effect of address space layout randomization can be eliminated. Our *signature normalization* will order the signatures based on the normalized virtual address.

(5) Other practical issues like page swapping and hot patch. There are other practical issues that complicate the signature checking problem. The kernel code pages may be swapped out. This is true for Windows OS but not for Linux (Linux kernel code cannot be swapped out [7]). Certain code areas may be hot patched by the third-party kernel modules. Hence, we need a *resilient signature matching scheme*.

To this end, we model the problem of signature matching as a string matching problem. Each element of our string is the MD5-hash value (with 32 bytes) of the identified pages. Some hash values may be missing due to page swapping, and some are different because of hot patching. Then we adopt the KMP algorithm [21] to find the closest match.

2.3 System Overview

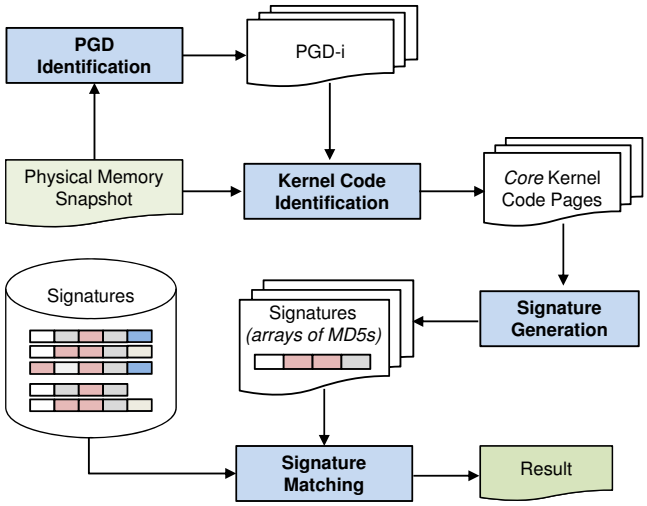

Figure 1: Overview of our OS-SOMMELIER

An overview of our OS-SOMMELIER is presented in Fig. 1. There are four key components in our system: (1) *PGD Identification*, (2) *Kernel Code Identification*, (3) *Signature Generation*, and (4) *Signature Matching*.

Given a physical memory snapshot, OS-SOMMELIER will first invoke the *PGD Identification* to search all the possible PGDs. Once correct PGDs are found, it 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 will also split the kernel code based on the virtual addresses and the internal caller-callee relation to identify the "*core*" kernel code pages.

Next, our *Signature Generation* will use our correlative disassembling technique to neutralize the side effect of code randomization, and then hash (MD5) each disassembled page and store it in an array based on the normalized virtual address of each hashed page. Finally, our *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.

For simplicity of the paper presentation, we focus our discussion on the widely used 32-bit x86 architecture. However, we believe our techniques can potentially be generalized and adapted to support 64-bit systems and even other CPU architectures.

3. DETAILED DESIGN

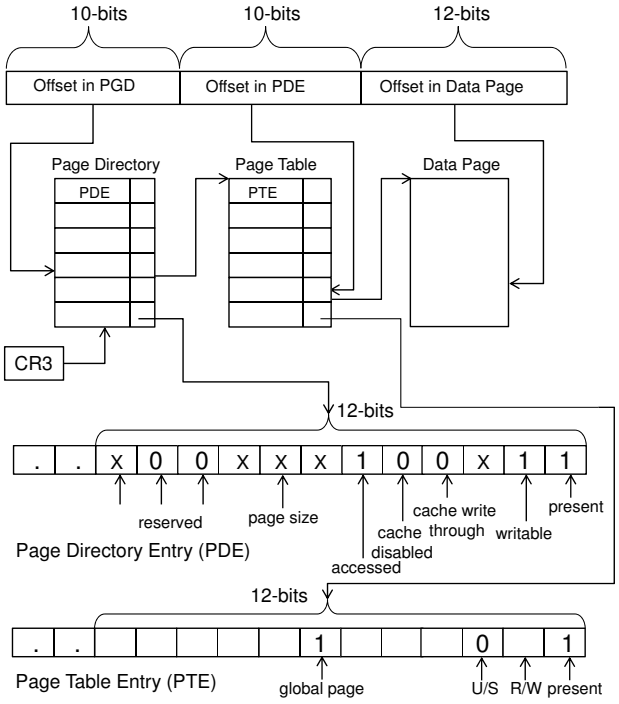

Figure 2: Address translation and the key PGD and PTE bit properties we exploited in 32-bits x86 architecture.

3.1 PGD Identification

To unveil the inherent characteristics of PGDs, we need to study the page table structure in the x86 architecture. As illustrated in Fig. 2, when translating a virtual address, the Memory Management Unit (MMU) first uses the left-most 10 bits of a virtual address as an index to look up the PGD table, which is normally pointed to by the control register CR3. The PGD table is an array of Page Directory Entry (PDE), which points to a page directory. The MMU then uses the middle 10-bits of a virtual address to look up the page directory table pointed to by the PDE entry. The Page Table Entry (PTE) contains the pointer which points to the final page frames containing the actual data.

Fundamentally, identifying PGDs is similar to the problem of data structure identification in memory forensics, because a page table is a three-level (points-to) table structure. Thus, we can leverage robust signature schemes to solve this problem. In particular, we make use of both field value-invariant [13] signatures in PGD entry and the strong field points-to signatures [23] (i.e., the points-to relationship between a PGD entry and the target page directory table). More concretely, for a valid PGD, there must exist at least one entry that points to a valid target page directory table, which is used for the kernel space mapping. For each entry in this page directory table, if the present bit (bit 0) is set, the system bit (bit 3) and the global bit (bit 7) should also be set. To speed up this search process, we can start from the middle of each page (i.e., the 512th entries) when verifying PGD entries. This is because for all OS implementations, the kernel space always occupies the upper half of the virtual memory space (e.g., `0x80000000-0xFFFFFFFF` for some versions of Windows and `0xC0000000-0xFFFFFFFF` for Linux).

The identified PGDs may be still noisy. To further filter out the erroneous PGDs, we perform consistency checks within these PGDs. As the kernel space is shared by all the running processes, the kernel portions of the identified PGDs should be extremely similar (if not completely identical). Therefore, we can immediately filter out the erroneous PGDs. The reason why all processes do not share the exact kernel space mapping is that kernel space could contain some process private data and the mapping for the private data will be different for different processes. In our experiment, we see processes share over 95% PGD entries in kernel space. Once all the possible PGDs in the physical memory are identified, we could use any one of them for virtual to physical address translation.

Also, there are other viable options to extract the PGD. For instance, cloud providers can directly extract the CR3 value in each running VM by modifying the hypervisor, and keep a copy of CR3 when saving the memory snapshot of a VM. In our design, we aim to enable OS-SOMMELIER to transparently analyze the memory snapshot without any other support. It therefore leads to the above signature-based approach in identifying the PGD.

3.2 Kernel Code Identification

Once we have identified the PGDs, the next step is to identify the possible "*core*" kernel code pages in the physical memory. There are three steps: (**Step-I**) we will perform the virtual to physical (`V2P`) address translation for kernel space by checking each PDE and PTE entry, 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 an identified PGD, we can correctly find the entire kernel space by checking the system bit in the page table entries. That is, without any prior knowledge, we are able to find out that the kernel space for Windows starts from `0x80000000` and the kernel space for Linux starts from `0xc0000000`.

Then we attempt to 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. There is an intuitive approach that we could narrow down the core kernel code by identifying the read-only kernel pages, because the kernel code should be protected as read-only. However, this approach is not general enough, as some legacy OSes may not enforce this code integrity protection. Actually, according to our experiment even Windows XP kernel code is not read-only. Its main kernel code and data are all allocated in a big 4MB page, which is both readable and writable. To be conservative, we do not take this approach.

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 possible solution would be to search for the page which contains instructions based on the code distributions [12], and such an approach has been actually used in unpacking (e.g., [33]) to differentiate code and data. However, this potential solution 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,

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

Table 1: X86 System Instruction Distributions in Kernel Code Pages

(2) have unique pattern (short sequence will have false positive), and (3) not in kernel modules.

According to the x86 instruction set [18], there are in total 28 system instructions and their instruction length and distributions in Linux, Windows and BSD UNIX 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 [18]. We confirm that this sequence (`0f 20 d8 0f 22 d8`) appears in all the kernels we tested.

Very often, 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 in direct function calls. 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 (e.g., Windows 7), 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 can invoke a direct function call (e.g., `printk`) into the main kernel, as its address can be statically resolved. 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 observation: the main kernel always occupies the lowest kernel space, leaving the higher kernel space 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. Based on this observation, we propose to explore the *direct forward function call* relation.

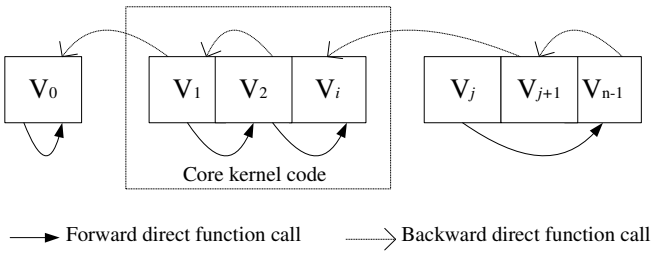

Figure 3: Illustration of core kernel code clustering.

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 3, by searching direct forward calls, we can exclude the device drivers.

Since we do not know in advance whether the OS kernel to be fingerprinted is relocatable or not, we choose the direct forward call based clustering approach for both relocatable and static kernels. This approach still works for the relocatable kernels, but the identified kernel code cluster might be smaller than actual, because the backward direct calls are omitted. For OS fingerprinting, this is acceptable because our goal is not to get full code coverage. 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 (§3.3).

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 ← new Cluster
3: C.start ← Vaddr(p_0)
4: C.end ← Vaddr(p_0) + 4096
5: C.page ← ∅
6: T ← ∅
7: for each p_i ∈ C_Kk do
8:     C.page ← C.page ∪ {p_i}
9:     for each DirectForwardFunCall f ∈ p_i do
10:        if FunTarget(f) > C.end then
11:            C.end ← FunTarget(f)
12:        end if
13:    end for
14:    if Vaddr(p_i) + 4096 > C.end then
15:        T ← T ∪ C
16:        C ← new Cluster
17:        C.start ← Vaddr(p_i)
18:        C.end ← Vaddr(p_i) + 4096
19:        C.page ← ∅
20:    end if
21: end for
22: T ← T ∪ C
23: C_CK ← T[0]
24: for each c ∈ T do
25:    if (TlbFlush ∈ c and |c| > |C_CK|) then
26:        C_CK ← c
27:    end if
28: end for
29: return C_CK
30: }
```

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 (line 16 - line 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.

3.3 Signature Generation

A naive signature generation scheme is to hash (MD5) each page in C_{CK} as the signatures. However, such an approach will fail for some modern OSes. For example, as shown in Fig. 4, Windows-7 actually randomizes the kernel instruction address, and during the randomization some code and data labels (boxed in the figure) for some instructions are changed as well.

Thus, we have to neutralize the randomization. To this end, we introduce a *code normalization* technique which distills the memory

```
0x828432b6: 33 f6                    xor esi, esi
0x828432b8: 83 3d 38 fe 99 82 02     cmp dword ptr ds[0x8299fe38], 0x2
0x828432bf: 0f 87 95 00 00 00        jnbe 0x8284335a
0x828432c5: 8b 0d 3c fe 99 82        mov ecx, dword ptr ds[0x8299fe3c]
0x828432cb: 33 c0                    xor eax, eax
...
0x82843432: e8 e4 9e 09 00           call 0x828dd31b

0x828182b6: 33 f6                    xor esi, esi
0x828182b8: 83 3d 38 4e 97 82 02     cmp dword ptr ds[0x82974e38], 0x2
0x828182bf: 0f 87 95 00 00 00        jnbe 0x8281835a
0x828182c5: 8b 0d 3c 4e 97 82        mov ecx, dword ptr ds[0x82974e3c]
0x828182cb: 33 c0                    xor eax, eax
...
0x82818432: e8 e4 9e 09 00           call 0x828b231b
```

Figure 4: Address pace randomization in Windows-7 Kernel.

and immediate operands and only hashes the opcode and register operand, based on a robust disassembly. In addition, we can also observe from Fig. 4 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 randomized kernels.

Correlative Disassembling Robust disassembling in general is a challenging task in x86, since code and data could be mixed, and code could start at any location (may not be aligned). In our OSSOMMELIER, we take a special and robust approach by exploring the constraint from the direct call instruction and the targeted function prologue.

More specifically, considering a direct call instruction `call 0xc108a8b0` (with the machine code `e8 25 2c 00 00`) shown in Fig 5(a) as an example, the operand of this instruction `25 2c 00 00` (0x2c25) is the displacement to the target callee address (0xc108a8b0), 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 indirect `jmp` instruction, we stop disassembling this function. In other words, our disassembling adopts a linear sweep algorithm [31]. We have tested this correlative disassembly approach with many binary programs including the OS kernel (to be presented in §4) and user level binary (c.f., our Bin-Carver [17]), and we did not encounter any false positives.

It is worth noting that the function prologue has several variants. Specifically, as depicted in Fig 5, the function prologue in Windows kernel always starts with `mov edi, edi` instruction, and such 2-byte instruction is mainly used for hot-patches and detouring (worked like a pseudo-NOP which can be on-line replaced with a short 2-byte `jmp` instruction) [25]. Also, the assembly code `mov ebp, esp` could have two different machine code representations, namely, `89 e5` in Linux and FreeBSD, and `8b ec` in Windows. Thus, in our function prologue checking, we will examine whether the machine code is `55 89 e5` (Linux/BSD UNIX) or `8b ff 55 8b ec` (Windows).

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., [22]).

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

(a) Linux Kernel

```
0x806eee0a:  e8 3d 69 00 00        call   0x806f574c
...
0x806f574c:  8b ff                 mov    edi, edi
0x806f574e:  55                    push   ebp
0x806f574f:  8b ec                 mov    ebp, esp
```

(b) Windows Kernel

```
0xc04d675f:  e8 0c cf 00 00        call   0xc04e3670
...
0xc04e3670:  55                    push   ebp
0xc04e3671:  89 e5                 mov    ebp,esp
```

(c) FreeBSD/OpenBSD/NetBSD Kernel

Figure 5: Exploring The Constraint between a caller instruction and the callee prologue for robust disassembling.

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.

Another consideration is needed for kernel page swapping. Some functions may not be disassembled because some pages are present at signature generation time, but not at signature checking time, or vice versa. Therefore, to stabilize the signature, we choose to only disassemble the callee code if the caller code is within the same page and incorporate the disassembled code into the signature. Because of this solution, the coverage of disassembled kernel code may decrease. We prefer to make a trade-off between the stability and the disassembly coverage, because again the disassembly coverage is not a critical factor for our fingerprinting problem.

Algorithm 2 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 ∈ C_CK do
3:     p_i.dis ← ∅
4:     for each DirectFunCall f in p_i do
5:         taddr ← FunTarget(f)
6:         if WithinPage(taddr, f) and Prologue(taddr) then
7:             p_i.dis ← p_i.dis ∪ {taddr mod 4096}
8:         end if
9:     end for
10:    data ← LinearSweepDisass(p_i)
11:    index ← Vaddr(p_i)-Vaddr(p_0)
12:    S[index] ← MD5(data))
13: end for
14: return S }
```

Our 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 (line 5 - line 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 (e.g., Windows kernel page may get swapped in a real run and in our signature training phase we will turn off kernel page swap option), 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 (line 11 - line 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.

3.4 Signature Matching

Finally, to use our system, as a one time effort, a cloud provider or a forensics examiner will have to first generate a database of signatures by collecting all the ground-truth using an array-organized MD5 representation for each of the OSes in their stable states. Subsequently, given a physical memory snapshot of any unknown OS, *PGD Identification, Kernel Code Identification*, and *Signature Generation* are performed to generate an array of MD5-signatures from the given memory snapshot.

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 [21] 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. EVALUATION

We have implemented our OS-SOMMELIER with 4.5K lines of C code and our *correlative disassembler* is built on top of XED [2] library. In this section, we present our experimental result. We first tested its effectiveness in §4.1, using over 45 OS kernels from five widely used OS families such as Microsoft Windows and Linux. Then we report the performance overhead of each component of our system with all the tested kernels in §4.2. Finally, we compare our result with the state-of-the-art host-based OS fingerprinting techniques in §4.3.

Experiment setup We evaluated a wide variety of OS kernels from Microsoft Windows, Linux, to BSD family (including FreeBSD, OpenBSD, and NetBSD). There are two phases of running our entire system:

OS-kernels	#PGD	$\|C_K\|$	$\|C_{Kk}\|$	#Pages	$\|T\|$	#Pages'	#DisPage	$P_r\%$	$D_r\%$	$\|S\|$	$\|S'\|$	$\sum$Time (seconds)
Win-XP	12	883	2	1024	16	384	232	60.42	11.91	232	1	0.544
Win-XP (SP2)	15	952	2	1024	13	421	277	65.80	15.18	277	1	0.6784
Win-XP (SP3)	15	851	2	1024	14	423	282	66.67	14.76	282	1	0.6606
Win-Vista	24	2310	1	1024	5	807	453	56.13	7.09	453	1	0.5818
Win-7	18	2011	2	280	1	280	178	63.57	5.75	178	1	0.522
Win-2003 Server	20	1028	2	1024	9	659	374	56.75	13.08	374	1	0.7676
Win-2003 Server (SP2)	19	1108	2	1024	6	563	342	60.75	12.66	342	1	0.7218
Win-2008 Server	20	1804	1	1024	9	849	542	63.84	7.71	542	2	0.6524
Win-2008 Server (SP2)	21	1969	1	1024	6	856	536	62.62	7.81	536	2	0.812
FreeBSD-7.4	27	369	1	3072	5	1758	1001	56.94	8.87	1001	1	1.3622
FreeBSD-8.0	20	350	1	3072	2	2959	1122	37.92	8.66	1122	1	1.4926
FreeBSD-8.2	27	360	1	3072	2	2956	1143	38.67	8.90	1143	1	1.5354
FreeBSD-8.3	18	412	1	4096	2	3966	1187	29.93	9.15	1187	1	1.6354
FreeBSD-9.0	21	360	1	4096	3	2281	1318	57.78	8.64	1318	1	1.6838
OpenBSD-4.7	20	187	1	1634	4	1631	1163	71.31	16.41	1163	1	1.9024
OpenBSD-4.8	12	833	1	1936	3	1934	1258	65.05	11.22	1258	1	1.6454
OpenBSD-4.9	8	834	1	1973	3	1971	1291	65.50	11.00	1291	1	1.665
OpenBSD-5	11	1106	1	1588	4	1585	1290	81.39	11.36	1290	1	1.6764
OpenBSD-5.1	7	1195	1	1596	3	1593	1293	81.17	11.00	1293	1	1.678
NetBSD-4.0	16	225	1	2006	3	1995	1069	53.58	7.81	1069	60	1.2532
NetBSD-4.0.1	11	220	1	2006	3	1995	1048	52.53	7.58	1048	60	1.2438
NetBSD-5.0	13	213	1	2048	11	1792	1148	64.06	8.43	1148	2	1.385
NetBSD-5.0.1	11	212	1	2048	11	1792	1147	64.01	8.28	1147	5	1.386
NetBSD-5.0.2	13	211	1	2048	13	1779	1138	63.97	8.45	1138	1	1.393
NetBSD-5.1	11	210	1	2048	8	1792	1182	65.96	8.28	1182	24	1.4256
NetBSD-5.1.2	13	210	1	2048	9	1792	1183	66.02	8.28	1183	24	1.4208
Linux-2.6.26	82	69	1	812	2	811	526	64.86	9.48	526	1	1.4514
Linux-2.6.27	77	72	1	845	2	844	548	64.93	9.57	548	1	1.4664
Linux-2.6.28	77	109	1	885	2	884	575	65.05	9.78	575	1	1.4884
Linux-2.6.28.1	77	31	1	1229	11	971	880	90.63	10.79	880	40	1.9248
Linux-2.6.28.2	74	31	1	1230	12	971	874	90.01	11.07	874	40	1.9502
Linux-2.6.29	79	106	1	908	2	907	597	65.82	9.76	597	1	1.5156
Linux-2.6.30	79	190	1	934	3	670	606	90.45	9.98	606	1	1.5734
Linux-2.6.31	84	26	1	1545	3	1098	976	88.89	10.99	976	2	2.1002
Linux-2.6.32.27	85	26	1	1589	2	1588	1005	63.29	11.33	1005	2	2.1686
Linux-2.6.33	87	28	1	1606	3	1152	1039	90.19	11.23	1039	2	2.1188
Linux-2.6.34	83	28	1	1617	2	1616	1043	64.54	11.19	1043	2	2.1304
Linux-2.6.35	96	28	1	1640	3	1175	1056	89.87	11.35	1056	1	2.1132
Linux-2.6.36	99	28	1	1641	3	1183	1071	90.53	11.15	1071	2	2.1128
Linux-2.6.36.1	78	36	1	1024	1	1023	926	90.52	11.29	926	5	2.0192
Linux-2.6.36.2	78	36	1	1024	1	1023	925	90.42	11.17	925	31	1.8254
Linux-2.6.36.3	76	36	1	1024	1	1023	930	90.91	11.15	930	31	1.8152
Linux-2.6.36.4	81	36	1	1024	1	1023	929	90.81	11.11	929	22	1.8272
Linux-3.0.0	58	149	2	1024	1	1023	975	95.31	15.55	975	1	2.2086
Linux-3.0.4	73	183	2	1024	1	1023	918	89.74	12.33	918	1	2.021
mean	43.24	481.57	1.17	1588.53	4.97	1351.57	879.91	69.75	10.41	879.91	8.5	1.5

Table 2: Detailed evaluation result of the first three components of OS-SOMMELIER for the tested OS kernels during the ground truth signature generation.

- **Ground-truth Collection Phase** To generate the ground-truth signature for each testing OS kernel, we used one physical memory dump. To ensure the signature quality, we disabled page swapping for each OS version, and thus all the kernel code pages were present in the memory. To obtain the physical memory dumps, we run each of the OSes in a QEMU [1] VM with 512M bytes RAM (131,072 pages with 4K bytes each). After the guest OS booted up, we took a memory dump to compute the ground truth hash values.

- **Testing Phase** For the deployment testing (i.e., to evaluate our signature matching in the cloud), 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 3, we also varied the VM configurations with different physical memory size (from 256M to 1G) 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 8G memory machine, installing a Ubuntu 11.04 with Linux kernel 2.6.38-8.

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

Table 3: Testing Memory Snapshot Configuration. Time is calculated right after the VM is power on.

4.1 Effectiveness

Table 2 presents the detailed experimental results for each tested OS. The results show the effectiveness of each individual component. More specifically,

PGD Identification As reported in the 2^{nd} column of Table 2, for each tested OS, our PGD Identification can identify on average 43.2 PGDs. We confirmed that these PGDs exactly matched with the running processes in the OS. In other words, there were no false positives and false negatives, thanks to the strong graph-based signatures [23] of the PGD items.

Kernel Code Identification As three steps are involved in kernel code identification, we report the output of these steps respectively. As presented in column $|C_K|$, we usually could identify 481.6 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 2 clusters (column $|C_{Kk}|$). The largest cluster in C_{Kk} contains on average 1588.5 kernel code pages, and this number is reported in the 5^{th} column.

Signature Generation We report seven categories (from the 6^{th} to 12^{th} column) of data for this component. In particular, in the 6^{th} column, we report the total number of clusters $|T|$ identified by connecting the pages with direct forward function call constraint; the 7^{th} 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 8^{th} column, and this ratio P_r reported in the 9^{th} 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 10^{th} 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 11^{th} column). Among all of these signatures, at least how many of the signatures $|S'|$ (reported in 12^{th} 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 4.97 clusters. The largest cluster contains on average 1351.57 pages. Among these pages, on average 879.91 of them can be disassembled with a ratio (P_r) of 69.75%. For each disassembled page, we covered 10.41% 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 879.91 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. Only NetBSD-4.x requires 60 pages hash comparison in the worst case.

To zoom-in why we have to perform at most $|S'|$ number of hash value comparison, we reported the total number of hash conflicts between these kernels in Table 4, Table 5, and Table 6 for each different category of the OS, respectively. If any two kernels are from different category, there is no conflict. Also, there is no page hash conflict for FreeBSD and OpenBSD. That is, any page hash could be used as a unique signature to fingerprint its kernel. For NetBSD-4.0 and NetBSD-4.0.1, there are 59 pages which have the same hash value as shown in Table 5.

Additionally, we did a manual verification on the identified largest core kernel code cluster, and we found the exact core kernel code for Win-XP whose kernel code is allocated in a 4M bytes page. For Win-7, it uses fine-grained kernel code pages, and our technique gets a more condensed kernel code. For BSD UNIX, it tends to have larger code size, and the identified core kernel code does belong to the original code.

Signature Matching With the generated signatures, we evaluate the correctness of efficiency of the signature matching process of OS-SOMMELIER. We used 5 physical memory dumps for each OS that we collected from the virtual machines with different configurations. We found that OS-SOMMELIER can correctly identify the right OS versions for all the memory dumps in our data set, regardless of the different configurations of the guest OS.

OS-kernels	Win-XP	Win-XP (SP2)	Win-XP (SP3)	Win-Vista	Win-7	Win-2003 Server	Win-2003 Server (SP2)	Win-2008 Server	Win-2008 Server (SP2)
Win-XP	-	0	0	0	0	0	0	0	0
Win-XP (SP2)	0	-	0	0	0	0	0	0	0
Win-XP (SP3)	0	0	-	0	0	0	0	0	0
Win-Vista	0	0	0	-	0	0	0	0	0
Win-7	0	0	0	0	-	0	0	0	0
Win-2003 Server	0	0	0	0	0	-	0	0	0
Win-2003 Server(SP2)	0	0	0	0	0	0	-	0	0
Win-2008 Server	0	0	0	0	0	0	0	-	1
Win-2008 Server(SP2)	0	0	0	0	0	0	0	1	-

Table 4: Summary of Page Hash Conflict for Windows Kernels

OS-kernels	NetBSD-4.0	NetBSD-4.0.1	NetBSD-5.0	NetBSD-5.0.1	NetBSD-5.0.2	NetBSD-5.1	NetBSD-5.1.2
NetBSD-4.0	-	59	0	0	0	0	0
NetBSD-4.0.1	59	-	0	0	0	0	0
NetBSD-5.0	0	0	-	1	0	1	1
NetBSD-5.0.1	0	0	1	-	0	4	4
NetBSD-5.0.2	0	0	0	0	-	0	0
NetBSD-5.1	0	0	1	4	0	-	23
NetBSD-5.1.2	0	0	1	4	0	23	-

Table 5: Summary of Page Hash Conflict for NetBSD Kernels

While our OS-SOMMELIER uses KMP string matching algorithm [21] 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 $|S'|$ column in Table 2), thanks to our strong, sensitive code hash based signatures.

4.2 Efficiency

We also measured the performance overhead of OS-SOMMELIER. The total signature generation time is presented in the last column of Table 2 for the testing kernel when we generate their ground truth signatures with 512M physical memory. For the real kernel test, we also have to go through this signature generation process. It is 1.50 seconds on average (with the worse case 2.21 seconds). For a fair comparison, this number does not include the physical memory file loading time, which is about 6.67 seconds. This is because for some application scenarios (such as virtual machine introspection), the live memory is already available for the fingerprinting.

To evaluate the efficiency, we fed the 5 memory snapshots for each of the testing kernels to OS-SOMMELIER, and it took on average an additional 0.03 seconds for the signature matching, and therefore in total given a physical memory dump, our system takes about 2 seconds in total to fingerprint an OS, which we believe is efficient enough for many application scenarios.

We further break down the total runtime into individual components, and list the results in Fig. 6. We can see that the biggest portion of the runtime is spent on the signature generation component, which takes on average 41% of the total time, because of the disassembling as well as the hashing procedure. The next one is the PGD Identification component which takes 32% of the time (to determine one physical page, we need to traverse many next

OS-kernels	Linux-2.6.26	Linux-2.6.27	Linux-2.6.28	Linux-2.6.28.1	Linux-2.6.28.2	Linux-2.6.29	Linux-2.6.30	Linux-2.6.31	Linux-2.6.32.27	Linux-2.6.33	Linux-2.6.34	Linux-2.6.35	Linux-2.6.36	Linux-2.6.36.1	Linux-2.6.36.2	Linux-2.6.36.3	Linux-2.6.36.4	Linux-3.0.0	Linux-3.0.4
Linux-2.6.26	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.27	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.28	0	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.28.1	0	0	0	-	39	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.28.2	0	0	0	39	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.29	0	0	0	0	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.30	0	0	0	0	0	0	-	0	0	0	0	0	0	0	0	0	0	0	0
Linux-2.6.31	0	0	0	0	0	0	0	-	1	0	0	0	0	0	0	0	0	0	0
Linux-2.6.32.27	0	0	0	0	0	0	0	1	-	0	0	0	0	0	0	0	0	0	0
Linux-2.6.33	0	0	0	0	0	0	0	0	0	-	1	0	1	0	0	0	0	0	0
Linux-2.6.34	0	0	0	0	0	0	0	0	0	1	-	0	1	0	0	0	0	0	0
Linux-2.6.35	0	0	0	0	0	0	0	0	0	0	0	-	0	0	0	0	0	0	0
Linux-2.6.36	0	0	0	0	0	0	0	0	0	1	1	0	-	0	0	0	0	0	0
Linux-2.6.36.1	0	0	0	0	0	0	0	0	0	0	0	0	0	-	4	4	4	0	0
Linux-2.6.36.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	30	21	0	0
Linux-2.6.36.3	0	0	0	0	0	0	0	0	0	0	0	0	0	4	30	-	21	0	0
Linux-2.6.36.4	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	21	-	0	0
Linux-3.0.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	0
Linux-3.0.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-

Table 6: Summary of Page Hash Conflict for Linux Kernels

layer pointer fields), and kernel code identification takes 25% of the time, because it has to compare any two contiguous PTE properties and perform a lightweight forward call search, verification, and clustering.

4.3 Comparison with Other Systems

Finally, we compared OS-SOMMELIER with the other two state-of-the-art host-based OS fingerprinting systems: UFO [30] and an IDT-based approach [10]. UFO explores the x86 CPU states, i.e., the register values of GDT, IDT, CS, CR, and TR, to fingerprint an OS. The IDT-based approach is similar to our system in that we both hash the OS kernel code for the fingerprinting, but the difference is that the IDT-based approach only hashes the interrupt handler pointed from the IDT registers. To evaluate these two systems, we implemented an IDT-based approach based on the description in the paper [10], and obtained the source code of UFO from the authors.

As summarized in Table 7, both UFO and the IDT-based approach are not accurate enough. UFO can identify all the FreeBSD/OpenBSD, and it also works for the majority of the Linux kernel we tested. However, it cannot correctly fingerprint several Windows versions. For the IDT-based approach, although it works well on Linux kernels, its accuracy on other OS families is unsatisfactory: it cannot pinpoint the differences between Win-XP, SP2 and SP3, and cannot differentiate the minor version in FreeBSD, OpenBSD and NetBSD. In contrast, our OS-SOMMELIER works perfectly for all the OS versions in our data set.

5. DISCUSSION

Given the threat model described in §2.1, in this section, we discuss the possibility of various evasion attacks and the limitations of OS-SOMMELIER.

Creating extra noisy data As 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. As a result, incorrect fingerprinting results will be reported based on the fake code pages.

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.

To create an inverse page for a MD5 tends to be impossible, and the fake page only introduces mis-matches with our database. Thus, we will eventually be able to recognize the true kernel code pages. It is also possible that an attacker can present kernel code pages obtained from a different OS kernel, in order to mislead OS-SOMMELIER. Such an attack would trick OS-SOMMELIER into reporting a fake OS version in addition to a real one. Then subsequent verifications can be done to invalidate the fake OS version. For example, virtual machine introspection can try each reported OS version to check if the extracted OS-level semantics is valid.

Obfuscating the kernel code If a VM user is uncooperative or malicious, she can obfuscate the kernel code 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 recompile the kernel with different options (e.g., omitting the stack frame pointer) if she has the source code.

In general, this kind of evasion attacks is equivalent to fingerprinting a completely new and unknown operating system, 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, if an OS is patched, OS-SOMMELIER may not be able to recognize it. In addition, we have not tested OS-SOMMELIER with any microkernels (e.g., MINIX). We leave it as one of our future works.

6. RELATED WORK

In general, fingerprinting an OS could be done from (1) network protocol perspective, (2) CPU-register perspective, (3) file system

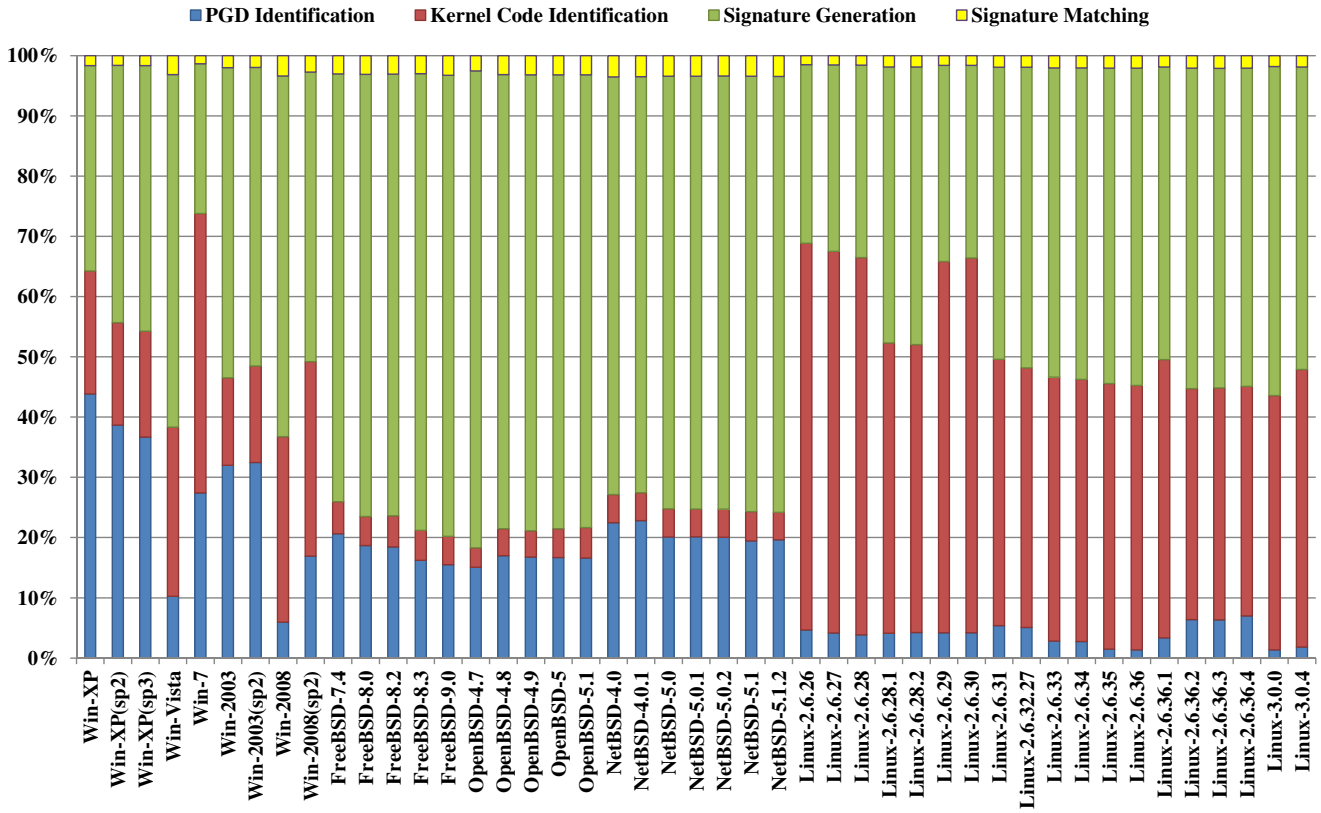

Figure 6: Normalized performance overhead for each component in OS-SOMMELIER**.**

perspective, and (4) memory perspective. In this section, we examine these related works and compare them with OS-SOMMELIER.

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 [26], 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 [15], Xprobe2 [3]/Xprobe2++ [42], and synscan [38], 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 [35].

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. For example, in many cases of memory forensics, the only input available is a memory dump. Also, as an enhanced security measure, many modern OSes (such as Windows Servers) disable most of the network services as a default security policy, and hence no responses can be observed.

Furthermore, a network based fingerprinting approach is not effective against anti-fingerprinting techniques (e.g., [29, 34]), which are used to defeat vulnerability scanner, penetration testing, and automated malware propagation. For an anti-fingerprinting protected machine, the accuracy will be quite limited while probing its OS-versions as the TCP/IP packets have been obfuscated.

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), as shown in our experiment 4.3 – 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 [20] already support this capability.

However, this approach is not feasible for a machine with encrypted file systems. For security and privacy concerns, it is very reasonable for cloud users to run their virtual machines with encrypted file systems. While it is possible for cloud providers to retrieve the cryptographic keys of the encrypted file system using

OS-Kernels	UFO	IDT-based	OS-Sommelier
Win-XP	✗	✗	✓
Win-XP (SP2)	✗	✗	✓
Win-XP (SP3)	✗	✗	✓
Win-Vista	✓	✓	✓
Win-7	✓	✓	✓
Win-2003 Server	✗	✓	✓
Win-2003 Server (SP2)	✗	✓	✓
Win-2008 Server	✓	✓	✓
Win-2008 Server (SP2)	✓	✓	✓
FreeBSD 7.4	✓	✗	✓
FreeBSD 8.0	✓	✓	✓
FreeBSD 8.2	✓	✗	✓
FreeBSD 8.3	✓	✓	✓
FreeBSD 9.0	✓	✓	✓
OpenBSD 4.7	✓	✗	✓
OpenBSD 4.8	✓	✗	✓
OpenBSD 4.9	✓	✗	✓
OpenBSD 5.0	✓	✓	✓
OpenBSD 5.1	✓	✗	✓
NetBSD 4.0	✗	✗	✓
NetBSD 4.0.1	✗	✗	✓
NetBSD 5.0	✓	✓	✓
NetBSD 5.0.1	✗	✗	✓
NetBSD 5.0.2	✗	✗	✓
NetBSD 5.1	✗	✗	✓
NetBSD 5.1.2	✗	✗	✓
Linux-2.6.26	✓	✓	✓
Linux-2.6.27	✓	✓	✓
Linux-2.6.28	✓	✓	✓
Linux-2.6.28.1	✓	✓	✓
Linux-2.6.28.2	✓	✓	✓
Linux-2.6.29	✓	✓	✓
Linux-2.6.30	✓	✓	✓
Linux-2.6.31	✓	✓	✓
Linux-2.6.32	✓	✓	✓
Linux-2.6.33	✓	✓	✓
Linux-2.6.34	✓	✓	✓
Linux-2.6.35	✓	✓	✓
Linux-2.6.36.1	✗	✓	✓
Linux-2.6.36.2	✗	✓	✓
Linux-2.6.36.3	✗	✓	✓
Linux-2.6.36.4	✓	✓	✓
Linux-3.0.0	✓	✓	✓
Linux-3.0.4	✓	✓	✓

Table 7: Comparison with other techniques.

such as VM introspection techniques, then the bottle neck for such approach will lie in how to reliably retrieve the keys. Note that a running VM may have multiple keys, e.g., some of them may be used for communications, and some of them maybe used for other encryption and decryption (besides the file system). 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 (as summarized in Table 7 in our experiment). 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 [23] 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 can also be used for OS fingerprinting [23]. However, SigGraph is not accurate enough to differentiate minor kernel versions, as the differences in kernel data structures in these two OS versions may be fairly small. In addition, SigGraph is far from being efficient, as it scans every pointer and non-pointer field and examines the data structures in many hierarchical levels.

7. CONCLUSION

We have presented the design, implementation, and evaluation of OS-SOMMELIER, a physical memory-only based system for precise and efficient OS fingerprinting in the cloud. The key idea is to compute the core kernel code hash 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. Our experimental result with over 45 OS kernels shows that OS-Sommelier can precisely fingerprint all the OSes without any false positives and false negatives within only 2 seconds on average.

Acknowledgment

We would like to thank the anonymous reviewers for the insightful comments to further improve our paper. We also thank Nguyen Anh Quynh for sharing his UFO source code with us. This work is supported in part by a research grant from VMware Inc, and by National Science Foundation under Grants #1018217 and #1054605. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the VMware and NSF.

References

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

[2] Xed: X86 encoder decoder. http://www.pintool.org/docs/24110/Xed/html/.

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

[4] 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 *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, Chicago, Illinois, USA, 2010. ACM.

[5] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Baltimore, MD, 2005. USENIX Association.

[7] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[8] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–, 2001.

[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 *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, Seattle, WA, USA, 2008. ACM.

[10] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security (CCSW '09)*, pages 97–102, Chicago, Illinois, USA, 2009. ACM.

[11] D. E. Comer and J. C. Lin. Probing tcp implementations. In *Proceedings of the USENIX Summer 1994 Technical Conference*, Boston, Massachusetts, 1994.

[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 *Proceedings of DFRWS 2010 Annual Conference*, 2010.

[13] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM.

[14] Y. Fu and Z. Lin. 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*, San Francisco, CA, May 2012.

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

[16] L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *Proceedings of the first USENIX Workshop on Offensive Technologies*, pages 1–10. USENIX Association, 2007.

[17] S. Hand, Z. Lin, G. Gu, and B. Thuraisingham. Bin-carver: Automatic recovery of binary executable files. In *Proceedings of the 12th Annual Digital Forensics Research Conference (DFRWS'12)*, Washington DC, August 2012.

[18] 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.

[19] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, Alexandria, VA, November 2007.

[20] R. W. Jones and M. Booth. Virt-inspector - display operating system version and other information about a virtual machine. http://libguestfs.org/virt-inspector.1.html.

[21] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[22] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, San Diego, CA, 2004.

[23] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. 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.

[24] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158. IEEE Computer Society, 2010.

[25] Hot Patching and Detouring, http://www.ragestorm.net/blogs/?p=17.

[26] V. Paxson. Automated packet trace analysis of tcp implementations. In *Proceedings of the ACM SIGCOMM*, pages 167–179, Cannes, France, 1997. ACM.

[27] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[28] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy*, pages 233–247, Oakland, CA, May 2008.

[29] G. Prigent, F. Vichot, and F. Harrouet. Ipmorph: fingerprinting spoofing unification. *J. Comput. Virol.*, 6:329–342, November 2010.

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

[31] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society, 2002.

[32] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, Stevenson, Washington, USA, 2007. ACM.

[33] M. Sharif, V. Yegneswaran, H. Saidi, and P. Porras. Eureka: A framework for enabling static analysis on malware. In *Proceedings of the 13th European Symposium on Research in Computer Security*, Malaga, Spain, October 2008. LNCS.

[34] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *Proceedings of the 9th Conference on USENIX Security Symposium*, Denver, Colorado, 2000. USENIX Association.

[35] C. Smith and P. Grundl. Know your enemy: Passive fingerprinting. identifying remote hosts without them knowing. *Technical report, Honeynet Project*, 2002.

[36] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conf.*, Las Vegas, Nevada, August 2006.

[37] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)*, pages 363–374, Chicago, Illinois, USA, 2011.

[38] G. Taleck. Synscan: Towards complete tcp/ip fingerprinting. In *Proceedings of the Canada Security West Conference (CanSecWest '04)*, Vancouver B.C., Canada, 2004.

[39] P. Team. Pax address space layout randomization (aslr). *http://pax.grsecurity.net/docs/aslr.txt*.

[40] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[41] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269. IEEE Computer Society, 2003.

[42] F. Yarochkin, O. Arkin, M. Kydyraliev, S.-Y. Dai, Y. Huang, and S.-Y. Kuo. Xprobe2++: Low volume remote network information gathering tool. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.

[43] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, Cascais, Portugal, 2011. ACM.