Operating System Fingerprinting for Virtual Machines

Nguyen Anh Quynh
Email: aquynh@gmail.com

Abstract

In computer security field, Operating System fingerprinting (OSF) is the process of identifying the OS variant and version. OSF is considered an important stage to decide security policy enforced on protected Virtual Machine (VM). OSF is also the first step of VM introspection process. Unfortunately, current OSF techniques suffer many problems, such as: they fail badly against modern Operating System (OS), they are slow, and only support limited OS-es and hypervisors.

This paper analyzes the drawbacks of current OSF approaches against VM, then introduces a novel method named *UFO* to fingerprint OS running inside VM. Our solution fixes all the above problems: Firstly, it can recognize all the available OS variants and (in lots of cases) exact OS versions with excellent accuracy, regardless of OS tweaking. Secondly, UFO is extremely fast. Last but not least, it is hypervisor-independent: we proved that by implementing UFO on Xen.

1 Introduction

OSF is the process of understanding which OS is running on a particular machine. OSF is helpful for the administrators to properly decide the security policy to protect their systems. For example, assume that we want to protect this machine against the Conficker worm [10]. If we know that this machine runs Linux, which is not exploitable by Conficker, we do not need secure it. But if that machine runs a specific version of Microsoft Windows, we have to look more closely: in case the OS is Windows 7, we can safely ignore the problem, because Conficker does not affect this Windows edition. But if the OS is Windows XP SP3, which can be remotely compromised by this worm, we should put a firewall around the machine to prevent the attack, and possibly IDS/IPS must be deployed on the network path to monitor the threat.

One more motivation for our research is that knowing the VM's OS is important for memory introspection on VM [6]: memory analyzing can only be done when we know exactly the OS version of the OS inside the VM, because every OS variant, and also OS version, is very different in the internal structure.

This research explores available methods to perform OSF on VM, to help the VM administrators to keep track of the OS-es installed inside their VMs. The reason is that in reality, each VM can be rented by different person, and runs whatever OS that the VM's owner setup inside. Even initially the administrator knows exactly the OS installed in a particular VM, which might not be true anymore after that, because the VM's owner can upgrade his OS anytime. This is the actual case with Windows OS: Windows XP users can upgrade their OS to Windows Vista, then to Windows 7 with ease. The other problem he must frequently cope with is to protect unknown VM, with unknown OS, which might be migrated into his physical machines anytime from the cloud.

Unfortunately, available OSF approaches against VM have many problems. Firstly, they fail badly against modern OS-es having none or minimal customization. Secondly, some methods are not as fast as we would desire: they might take at least few dozen seconds for one target. Lastly, some methods depend on the hypervisors, and can only recognize particular OS-es.

We propose a novel method named *UFO* to solve these outstanding issues: UFO can recognize all the available OS-es variants, and even their exact versions, with excellent accuracy. We perform fuzzing fingerprints, so UFO can also deal with OS having non-trivial customization. The other benefit is that UFO is extremely fast: the fingerprinting is done in in milliseconds. Besides, it supports all kind of hypervisors. We proved that by implementations for Xen [14] and Microsoft Hyper-V [9].

2 Available OSF Solutions for VM

This section discusses the current problems of available OSF solutions for VM, then proposes several requirements for a desired OSF tool for VM.

2.1 Network-based OSF

Many OSF tools, such as *nmap* [5] and *xprobe* [15], have been introduced to actively perform remote OSF via network. While these tools are traditionally used against physical systems, they can be perfectly employed to fingerprint VMs exposed to the network, too.

However, all the network-based OSF methods suffer some major problems: they either rely on scanning open ports on the target, or on examining the replied packets. Unfortunately, nowadays modern systems tighten their setup, thus remote OSF fails in many cases. For example, Windows 7 disables all the network services by default, therefore close all the TCP/UDP ports. As a result, nmap cannot find any open ports, thus have no information to perform fingerprinting. The on-by-default firewall on Windows 7 also drops all the ICMP packets, so leaves no chance for xprobe, who relies on ICMP data to work.

The other problem is that network-based OSF is quite slow: nmap, a tool having a lot of optimization, typically takes at least 30 seconds on one target, even when we run nmap on the host VM, against a guest VM on the same physical machine. This problem is unavoidable, because nmap has to scan thousand ports on the target, then must wait for the responses, with specific timeout.

Last but not least: it is a trend that the administrators are more aware of OSF issue, and start to deploy anti-finger OS solutions, such as *ipmorph* [12], in their system to fool all the current network-based OSF methods. The fact that the tool like ipmorph is free and easy-to-use renders network-based OSF obsolete in many cases.

2.2 Memory Introspection

Memory introspection is the method of inspecting and analyzing the raw memory of the guest VM from the host VM, to understand the context and status of the guest [6]. This technique can disclose unlimited information about the guest OS, including even OS version and options. While memory introspection sounds like a solution for our problem, however memory introspection must be done in the other way around: we should know exactly the OS runs inside the guest first to apply the right introspection method to analyze its memory. Unfortunately, there are a lot of OS-es to be recognized, and even with the OS source code in hand, understanding the OS internals to extract the desired information is far from trivial.

Paper [4] proposed an interesting method to fingerprint the OS without having to know the OS internals, that is to compare the hash value of the first code fragments of the interrupt handlers with known OS-es. The authors claimed that this is possible because the interrupt handlers vary significantly across OS types and versions. However, this idea misses an important point: the binary code depends on compilers, compiler versions, and also compiler settings used to compile the OS. For open source OS, such as Linux, *BSD or OpenSolaris, the kernel can be recompiled by users using whatever compiler and compiler options they want to. In such a case, even with the same source code, the interrupt code might vary accordingly, and the hash value of the interrupt handler greatly change. Consequently, the method of [4] fails to recognize the OS, even if the OS internals are unchanged.

2.3 Inspecting File-system Content

Another solution is to mount the guest file-system (FS), extract out special files and analyze their contents for information on OS variant and version. This approach is feasible because in principle, the host VM can access to the guest's FS, and reads its content. This method is already used by a tool named *virt-inspector* [8]: it mounts the FS, then extracts out and analyzes the registry files for Windows version [7].

However, there are some significant problems with this approach. Firstly, if the guest uses unknown FS, it is impossible for the host to mount its FS and access to its content. For this exact reason, the solution is not really portable to non-Unix hypervisors, such as Microsoft Hyper-V, with the host VM is based on Windows OS. Indeed, currently Windows can only understand *ext2* FS [1], but fails to recognize various other important FS-es in Linux world, which can be used in Linux-based VM running on Hyper-V.

Secondly, in case the guest encrypts the FS (which is a reasonable way to provide some security and privacy for guest VM in cloud environment), it is practically impossible for the host to understand the FS content.

Bottom line, we can see that all the available solutions examined above are not quite capable to solve the OSF problem for VM. We desire a better OSF tool with the following six requirements: (1) It gives accurate fingerprint result, with details on the target OS version. (2) It does not depend on the compiler using to compile the OS. (3) It is more resilient against OS tweaking. (4) It is not easy to be fooled by currently available anti-OSF tools. (5) It is faster, and should not cause any negative impact on the guest performance. (6) It can work with all kind of hypervisors. On the other words, it should be hypervisor independent.

Our paper tries to solve the OSF problems against

VMs running on the Intel platform, the most popular architecture nowadays.

3 UFO Design

Within the scope of this paper, we put a restriction on UFO: UFO does not try to fingerprint the real-mode OS-es. We just simply report that the guest VM is operating in real-mode if that is the case, without trying to dig further. While this is a limitation of UFO, it is not a big problem in reality, because most, if not all, modern OS-es mainly function in protected mode to take advantage of various features offered by Intel architecture.

3.1 Intel Protected Mode

Protected mode is an operational mode of Intel compatible CPU, allowing system software to utilize features not available in the obsolete real-mode, such as virtual memory, paging, etc..

Intel organizes system memory into segments, allowing the OS to divide memory into logical blocks, placing in different memory regions. In protected mode, each segment is represented by *segment selector*, *segment base* and *segment limit*. The segment selectors are represented by six segment registers *CS*, *DS*, *ES*, *FS*, *GS* and *SS*. All the segment information is stored inside a table called *Global Descriptor Table* (*GDT*). The base and limit of GDT are kept in *GDTR* register. When switching from real-mode to protected mode, OS must setup the GDT, using the *LGDT* instruction.

Protected mode OS also needs to initialize the *Interrupt Descriptor Table* (*IDT*), where put all the interrupt handlers of the system. The position and limit of IDT are kept in a register name *IDTR*, and IDT must be setup by the *LIDT* instruction.

OS can manage its tasks with a register named *Task Register* (*TR*). TR contains the segment selector of the Task-State-Segment (*TSS*), where kept all the processor state information of the current task. TR points to the GDT, thus can also be represented by segment selector, segment base and segment limit.

To provide strong isolation between privilege execution domains, Intel defines *four rings* of privilege: ring *0*, ring *1*, ring *2* and ring *3* (Typically, the OS kernel runs at ring 0, and applications run at ring 3). At any moment, the machine is functioning in only one of these rings.

An OS might use some special registers specific to hardware, but supported in all the modern CPU, like *MSR EFER* to control various features of the CPU, such as *64-bit OS* (to support 64-bit mode), *fast-syscall* (to enable faster execution system call, using modern instructions such as *SYSENTER* and *SYSCALL*), and *non-executable* (also called *NX* in short, to allow marking of memory pages as non-executable to prevent execution of malicious data placed into stack or heap by an attacker). Each of these features must be enabled by writing to the MSR EFER with an instruction named *WRMSR*. Therefore we can read the value of MSR-EFER to know if the OS supports these features or not.

Note that while 64-bit, fast-syscall and NX features have been introduced for quite a long time, for a lot of reasons, many OS-es have not supported them yet, or just picked them up in recent versions.

3.2 OS Parameters

From the external point of view, an OS uses several facilities, making a set of *OS parameters*, defined as followings.

- **Segment parameters**: each of six segment registers CS, DS, ES, FS, GS and SS is considered an OS parameter. These *segment parameters* have following three attributes: *segment selector*, *segment base* and *segment limit*, represented the selector, the base and the limit of the segment, respectively.

  Because at a moment, the machine is operating at one of four ring levels of privilege, we need to clarify the privilege of each segment parameter. We associate them with the ring level they are functioning in. For four ring levels, potentially with each segment register we can have up to four possible segment parameters. For example, with code segment CS, we have *CS0*, *CS1*, *CS2*, *CS3* parameters, respectively for ring 0, ring 1, ring 2, and ring 3. Similarly, we have four set of segment parameters for each of remaining segment registers DS, ES, FS, GS and SS.

- **TR parameter**. The task register TR refers to a segment, so similarly to segment registers above, it consists of *segment selector*, and *segment limit* attributes, represented the selector and limit of the TSS segment, respectively.

  Note that unlike segment registers, we do not consider the segment base as an attribute of TR, because the TSS can locate anywhere in the memory, thus its base does not represent the OS character. Our experiments with various OS-es confirm this fact.

- **GDT parameter**: The GDT can be located by the its base and limit. However, similar to TR above, we ignore the GDT base, and simplify this parameter by having the limit as its only attribute.

- **IDT parameter**: Similarly to GDT, the *IDT parameter* has IDT limit as its only attribute.

- **Feature parameters**: we consider each of the following OS features a *feature parameter*: *64-bit* (reflecting that the OS is 64-bit[1]), *fast-syscall* (reflecting that the OS uses fast-syscall facility), and *NX* (reflecting that the OS uses non-executable facility). When present, these parameters reflect that the corresponding facilities are supported by the OS.

All these OS parameters of each guest VM can be retrieved from the VM's context, thanks to the interface provided by the hypervisor, usually come in the shape of some APIs. These APIs can be executed from the host VM.

3.3 UFO Fingerprinting Method

We observe that to some extent, the protected mode of Intel platform enforces no constraint on how the OS is implemented, so the developers can freely design their OS to their desire. Our UFO method relies on the fact that most, if not all, modern OS-es spend very little time in real-mode after booting up, then they all quickly switch to protected mode, and mostly stay in this mode until shutdown. After entering the protected mode, each OS has different way to setup its low level facilities, such as GDT and IDT table, how it uses its registers, and whether or not it supports modern features like 64-bit, fast-syscall, and NX. Indeed, we can see that the limits of GDT and IDT tables, the value of segment registers and special registers, like TR, are significantly different between OS variants, and sometimes even between versions of the same OS. On the other words, each OS has different OS parameters, defined above.

Below are several cases on how some OS-es setup their OS parameters:

- Windows OS uses a GDT with the limit of *0x3FF*, while Linux *2.6* kernel has a GDT's limit of *0xFF*. Minix setups its IDT's limit of *0x3BF*, but Plan9's IDT has the limit of *0x7FF*.

- Sun Solaris uses selectors *0x158* and *0x16B* for its ring 0 and ring 3 code segments, respectively. Meanwhile, Haiku uses selectors *0x8* and *0x1B* for its *ring 0* and *ring 3* code segments.

- The 32-bit version of OpenBSD does not use the full address range for data segment like other OS-es, but dedicates the last part of 4GB address space for trapping security exploitation in W-xor-X technique. Therefore, its data segment of ring 3, represented by *DS3* parameter, has the limit of *0xCFBFDFFF*, rather than usual *0xFFFFFFFF*.

[1]Without this feature, the OS is in 32-bit mode

- Neither NetBSD nor FreeBSD uses fast-syscall feature. Meanwhile Linux started to use that from *2.6* kernel version, and Windows only started to take advantage of this feature from *Windows XP*.

- Windows only started to use NX feature from *XP-SP2*. All the prior versions did not take advantage of this modern facility.

The list of examples can go on, and it shows that the OS parameters can represent an OS. Based on these deviation of the OS parameters, we can recognize the OS variants, and even exact OS versions in various cases. Because the OS parameters can identify the OS, we consider the set of all parameters of an OS its *OS signature*, or *signature* in short.

To perform fingerprinting, we have to prepare signatures for all the OS-es we want to identify. We generate a signature for each OS, and put them into a database of signatures, called *signature database*. Then at run-time, we retrieve the OS parameters from the target VM, using hypervisor API discussed above, and match them against each signature in the signature database, as followings: we match each VM parameter against a corresponding parameter in the signature. One OS parameter matches the corresponding signature parameter if all of their attributes match each other.

A special case must be handled regarding the segment parameters: for each segment parameter, the corresponding signature parameter is the segment parameter of the ring level that the VM segment parameter is functioning in. For example, with code segment parameter CS, if the VM is operating in ring 0 at the time the OS parameters are retrieved, we have to match it against the CS0 parameter in the OS signature. To be matched, all attributes (ie. segment selector, segment base and segment limit) of the VM's CS parameter must match with corresponding attributes of CS0 parameter in the signature.

Regarding the feature parameters (64-bit, fast-syscall and NX) of the VM, a parameter is considered matched if it contains the list of features of the signature.

Ideally, the fingerprinting process would return the exact OS as the result. However, in fact we might not always find the definite answer, either due to missing signature of the related OS, or the OS has some special customizations deviating its parameters from its signature. To deal with this problem, we propose a fuzzy fingerprinting method: for each VM parameter matching the corresponding parameter in a signature, we give it *1* point. Otherwise, we give it *0* point.

We conclude the matching process for each signature by summing up all the points, and consider that the score of this signature. We repeat the matching with all the signatures in the database, and the signatures having the highest score will be reported as the potential OS of the VM.

We can see that in case we have the exact signature of the VM, UFO will find that and report it as the *100%* matched signature, thus identify the 100% correct OS, corresponding to that signature. More than one signature that has 100% match score is possible, and all will be reported.

In case the OS signature is either unavailable, or the OS is customized to a particular extent, so that its signature is not 100% matched any more, this fuzzy matching method can give a good guess on the OS by reporting the best matched OS-es with highest scores.

The design of UFO solves the outstanding problems of current OSF methods, and satisfy all of our six requirements proposed in section 2 above: (1) UFO can give a very accurate answer on the VM's OS. Experiments shows that UFO always report the OS variant correctly. The OS is also identified with details on OS version. (2) We never rely on the content of OS code to recognize the OS, so our method is independent of the compiler using to compile the OS. (3) Normal OS tweaking does not change how OS uses its OS parameters, and in general, OS has no option to affect their values, either. (4) Typically OS has no option to change the way it setups the low level facilities such as OS parameters. Consequently, while it is not impossible to modify these parameters, it is not trivial to fool UFO. Indeed, all the available anti-OSF solution do not work against UFO. (5) Because it takes a very little time to retrieve OS parameters from the VM and match them against the signature database, UFO is extremely fast. Moreover, we never create any network traffic like in network-based OSF approach, thus avoiding the possible problem of wasting the network bandwidth. (6) The way we generate the signatures and use them to perform matching does not depend on the hypervisors. Consequently, the same method and signature database can be used for all kind of hypervisors, as long as they provide the interface to retrieve the OS parameters of the guest VM at run-time.

3.4 Generate OS signatures

An OS signature must include all the values of its parameters. A naive approach to gather all the possible parameters is to have a tool running in host VM, and this tool periodically queries the targeted VM for its OS parameters. However, this solution has a major flaw: it is not guaranteed that such an external tool can collect all the possible values of all OS parameters, especially because some parameter values only appear at a particular moment, in a particular condition, and in a very short time. Even we can reduce the time interval of taking snapshot at the guest VM, we can never be sure that no event is missed.

To solve the above problem, we developed a method to guarantee that we do not miss any OS parameters, as long as it happens during the profiling time: We run the target OS in a special system emulator, which is instrumented at the right places to inform us when the OS starts to perform an activity that can generate a new value of a particular OS parameter. Our emulator then captures the new value, and records it for later processing. This profiling process is done from when the OS boots up, enters and processes in normal operating stage, until it is shutdown later.

After the OS is shutdown, the instrumented emulator processes all the collected OS parameters, then automatically generates the signature for the OS. This profiling process is repeated for all the OS variants and OS versions that we want UFO to be able to recognize, and all the generated signatures are put into the signature database.

This "dummy" method of profiling OS to create its signature offers a major advantage: it does not depend on prior knowledge on particular OS, and it should work blindly against all kind of OS-es, while requiring no understanding about the OS internals.

Because the OS behaves in the same way under different hypervisors, our signatures are hypervisor-independent, and can be reused by all the hypervisors.

4 Implementation

To generate the OS signatures, we have a tool to named *UFO-profiler*. We implemented UFO-profiler based on the QEMU emulator, version 0.11.1 [3]. QEMU is suitable because it supports all the facilities and features such as 64-bit, fast-syscall and NX, like other hypervisors. UFO-profiler instruments QEMU to record every time the related OS parameters change their values. The OS is run in UFO-profiler, and the profiling process starts when the VM switches to protected mode, and ends when the VM shutdowns. During the profiling, we run several usual applications in the VM to simulate the production systems, so we can trigger as much execution paths of the OS as possible. The output data is then processed to produce the ready-to-use signature.

On Intel platform, a lot of instructions and system events can change the OS parameters, either by directly modifying the OS parameters, or indirectly influencing their values. Specifically, the following Intel instructions are instrumented by UFO-profiler: *mov*, *int*, *sysenter*, *sysexit*, *syscall*, *sysret*, *jmp far*, *call far*, *lds*, *les*, *lfs*, *lgs*, *lgdt*, *lidt*, *ltr*, *wrmsr*, *loadall*, and *iret*.

Besides the above instructions, UFO-profiler must also instrument to monitor some low level activities that modifies OS parameters. The interested events are task switching and interrupt handling: the task switching involves a lot of operations that affect OS parameters, such as loading the new selector into TR register, switch to new segments in another privilege levels. Meanwhile, the interrupt handling process leads to new values are loaded into segment registers such as CS and SS.

QEMU dynamically translates all the instructions and events, so UFO-profiler simply instruments at the right places, when above instructions and events happen. At that time, the new value of related OS parameters is record, with all of its attributes. With all the saved parameters, we also record the time of the event, so we know the relative time of every events since when we start profiling the OS.

After the OS in the emulator is shutdown, the signature can be generated. It contains all the values of OS parameters we saved, with repeated values eliminated. The signature includes all the captured values of OS parameters, except the segment parameters: unlike other parameters, segment parameters might have a lot of values, which results in a big signature. The reason is that some segments might switch to different segment base, or set different limits for different purposes. We reduce the size of the segment parameters by *combining* them, if the segment base or limit attributes of all the parameters share the same *postfix*. In that case, the postfix is used to represent the attribute. For example, a base of *0x3F0BC0* and a base of *0X412BC0* share the postfix of *0XBC0*. Combining segment attributes has another benefit: the postifx can also match the segment attributes we do not encounter at profiling time.

When profiling various OS-es to generate their signatures, we observed that every OS must go through multiple steps of setup, until it reaches the *stable stage*. We simply define the ”stable stage” the time when OS is already in normal operation mode, and from there it does not change the setup of OS parameters such as GDT, IDT and MSR-EFER anymore, or only change them for its internal function, in special cases. Before reaching stable stage, the OS is in *unstable stage*, in which they may temporarily change the above parameters multiple times. Typically, one OS takes very little time (5 seconds in most cases, or no more than 20 seconds in all of our experiments) to reach the stable stage. More importantly, most of the time we fingerprint the OS, it is already in stable stage. Therefore, including the OS parameters collected at the unstable stage in the signature would create a lot of negative noises. This might make our fingerprinting result less accurate, because UFO can make mistake by using unstable parameters of other OS when considering all the matching results.

For this reason, we separate the signature into two parts: one includes only parameters of stable stage, and the other includes the parameters of unstable stage. When matching the OS parameters against the signature, we report both results of matching stable and unstable parameters. The users can decide what is the good result, because they might know which stage the OS is in at that time. (For example, all the virtual machine management tools let administrator know how long their VM already run).

We distinguish the parameters of stable and unstable stage thanks to the time recorded with all the parameters, as presented above. We simply define that parameters recorded after 60 seconds from when we start profiling the OS are stable, and all the parameters happened before that are of unstable stage. The time of 60 seconds is chosen based on our experiments, which makes sure that all the OS-es when profiling are already in stable mode at that time. This number does not really matter, as long as no stable parameters are missed during profiling.

The signature is presented in a JSON-like format [2], so it is human-friendly, easy to understand and modify. The database includes all the signatures in text format. Figure 1 in the appendix shows a signature of the 32-bit Linux kernel, version 2.6.31.x.

To prove that UFO is independent of hypervisors, we implemented it for two hypervisors: Xen and Hyper-V. The only difference between these two implementations is the way to retrieve the OS parameters from guest VM. Both Xen and Hyper-V provide interface to retrieve the OS parameters ([11], [13]). UFO runs inside the host VM, and retrieves the OS parameters from the targeted VM via an appropriate hypervisor interface. It matches these parameters against signatures using the matching algorithm proposed in the section 3.3 above. A special case is the feature parameters: UFO extracts the features from the MSR-EFER register of the VM.

5 Evaluation

Evaluation against various OS-es available shows that UFO is extremely fast: it takes around only 20 milliseconds to fingerprint the VM, regardless of OS. The time spent includes the time to retrieve the OS parameter values from the VM, parsing the signature database, loading them into the memory to prepare for matching step, and match the OS parameters against all the signatures available in the memory.

So far, the signature database of contains 23 OS variants, ranging from popular OS such as Windows, Linux, *BSD, Solaris, Plan9, Haiku,... to hobby and research OS-es such as Syllabale, Aros, Minix, ReactOS, Plan9, etc... A lot of versions of these OS-es are supported, too. Totally, UFO succesfully recognizes around 50 OS versions.

In our experiments, UFO identifies the OS variant with 100% accuracy. If it cannot give out the exact OS version, UFO can report a range of versions for the answer. For example, because Linux kernels from version 2.6.22 to 2.6.29 are similar in their signatures, this range of versions will be reported.

In the case UFO cannot identify the OS due to the missing signature, it can still give out the best match OSes, which has the highest score. In all of our experiments, the answer is always the closest OS versions. For example, UFO reports Windows Vista as an OS that best matches with Windows XP-SP3. This is because Vista matches 11 out of 12 parameters of Windows XP-SP3 signature, thus it is *91.66%* similar to Windows XP-SP3.

6 Discussion

This paper handles the security threat as in cloud enviroment: we assume that the hypervisor is secure, and cannot be breached from inside the guest VM. Meanwhile, the attacker might completely controls the guest, and can employ some tricks to defeat UFO to make the fingerprinting result incorrect. For example, even if the OS does not support NX feature, he can dynamically manipulate the kernel to enable the flag in MSR EFER to fool UFO. Ultimately, for the OS with source code available, he can modify the code, recompiles then replace the kernel to completely change all the OS parameters.

For the small modifications to the OS, UFO can still give a hint about the OS thanks to its fuzzy detection, which is close in many cases. However, it can be completely fooled on large scale modification. Unfortunately, this problem is hard to fix due to the constraint that we can only rely on the limited information collected from outside, with available interface.

The other problem is that UFO is not always able to give out the exact version of the OS: this happens because many OS versions do not change their parameters, like in the Linux case above. We can fix this problem by combining it with memory introspection method. In principle, we can use UFO in preliminary phase to recognize the OS variant, then confirm the exact version with memory introspection. The later phase requires knowledge about the OS internals, obviously.

7 Conclusions

UFO is a novel method of fingerprinting OS running inside VM. Being designed specifically for VM environment, UFO can quickly identify the OS variants and OS versions with excellent accuracy. The approach is hypervisor-independent, and the OS signatures of UFO can be reused for all kind of hypervisors.

References

[1] Ext2 installable file system for windows. `http://www.fs-driver.org`.

[2] JSON scheme format. `http://www.json.org`.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, 2005.

[4] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security. In *ACM workshop on cloud computing security (CCSW)*, 2009.

[5] Fyodor. nmap - free security scanner for network exploration and security audits. `http://nmap.org`.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[7] R. W. Jones. Explore the windows registry with libguestfs. `http://rwmj.wordpress.com/2009/06/08/explore-the-windows-registry-with-libguestfs`.

[8] R. W. Jones. virt-inspector. `http://libguestfs.org/virt-inspector.1.html`.

[9] Microsoft Corp. Windows server 2008 r2: Virtualization with hyper-v. `http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx`.

[10] Microsoft Support. Virus alert about win32/conficker worm. `http://support.microsoft.com/kb/962007`.

[11] MSDN. Windows driver kit: Hypervisor hypervisor c-language functions. `http://msdn.microsoft.com/en-us/library/bb969818.aspx`.

[12] G. Prigent, F. Vichot, , and F. Harrouet. Ipmorph : Fingerprinting spoofing unification. In *Proc. SSTIC*, June 2009.

[13] Xen project. Xen interface. `http://www.xen.org/files/xen_interface.pdf`.

[14] Xen project. Xen virtual machine monitor. `http://www.xen.org`.

[15] 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. International Conference on Dependable Systems and Networks*, July 2009.

Appendix

An OS signature is put inside a pair of opening and closing parentheses. The numbers are in hexadecimal mode, and each line denotes an OS parameter. All the text on the same line, after the sharp mark (*#*) is comment, and will be ignored. Each parameter might have multiple values, all is put inside a pair of square brackets. For those parameters having more than one attribute (like segment parameters, or TR), all the attributes of a value are also put inside another pair of square brackets. The attributes of segment parameters are in the order of *selector*, *base* and *limit*. The attributes of TR parameters are in the order of *selector* and *limit*. The postfix of segment attributes is represented with a '*' letter: in below example, segment GS in ring 3 (*gs3* parameter) has the base of modulo 16, represented by the postfix **0*.

```
{                      # Begin a signature with '{'
  name: Linux,         # OS name
  version: 2.6.31.x,   # OS version
  stable: {            # Begin of stable parameters
    idt:    [7ff],
    gdt:    [ff],
    tr:     [[80, 206b]],
    cs0:    [[60, 0, ffffffff]],
    cs3:    [[73, 0, ffffffff]],
    ds0:    [[0, 0, ffffffff]],
    ds3:    [[7b, 0, ffffffff]],
    es0:    [[0, 0, ffffffff]],
    es3:    [[7b, 0, ffffffff]],
    fs0:    [[d8, *, ffffffff], [0, 0, 0]],
    gs3:    [[33, *0, ffffffff]],
    ss0:    [[68, 0, ffffffff]],
    ss3:    [[7b, 0, ffffffff]],
    features: [nx, fast-syscall]
  },                    # End of stable parameters
  unstable: {           # Begin of unstable parameters
    idt:    [0, 3ff],
    gdt:    [30, 27, 1f],
    cs0:    [[10, 0, ffffffff]],
    ss0:    [[18, 0, ffffffff], [0, 0, ffffffff]]
  }                     # End of unstable parameters
}                       # End a signature with '}'
```

Figure 1: The signature of 32-bit Linux kernel, version 2.6.31.x