Image-Based Kernel Fingerprinting

By

Vassil Roussev, Irfan Ahmed and Thomas Sires

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 USA

Denver, CO (Aug 3rd - 6th)

Digital Investigation 11 (2014) S13–S21

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Image-based kernel fingerprinting

Vassil Roussev[*], Irfan Ahmed, Thomas Sires

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Digital forensics
Memory analysis
Operating system fingerprinting
Approximate matching
sdhash

ABSTRACT

The correct identification of operating system kernel versions is the first critical step in deep memory analysis—it enables the precise parsing of the kernel data structures and the correct interpretation of the observed system state. Identifying the exact kernel version is particularly challenging for open source operating systems where kernel upgrades are released frequently, and custom versions can be created on demand. State of the practice approaches, such as *Volatility*'s, rely on small and fragile signatures; state of the art research work relies on intricate understanding of architecture-specific implementation details, which limits them to Intel x86 environments, and requires continuous updates to identify the distinguishing characteristics of new kernels.

In contrast, our work builds robust signatures based solely on the content of the kernel images on disk, and is able to efficiently distinguish among incremental kernel version updates. The approach is entirely content-driven and requires no low-level analysis of the operation of the kernel. It utilizes an approximate matching tool—*sdhash*—to extract kernel fingerprints, and can be applied across different architectures without the need to parse and interpret the RAM snapshot. In addition, our evaluation data which contains hundreds of kernels, provides insights into the typical levels of content similarity across related kernels.

Introduction

Identifying the precise (operating system) kernel version is critical to both memory/kernel dump forensics, as well as VM introspection, management, vulnerability identification, and pentesting applications. In a cloud environment, the service provider needs to know the specifics of the deployed stack in order to provide system and network security services, monitor the deployed VMs for abnormal behavior, and maintain an accurate overall picture of the VM population.

Despite some common concerns, such as *precision*, the specific requirement of security and forensic applications have somewhat different points of view on the kernel fingerprinting method requirements. In the VM monitoring/management scenario, tenant *privacy* and overall (throughput) *performance* emerge as important problems. Tenants would strongly prefer a provider who does not continuously trawl through their VM file systems, and the provider itself would not want to dedicate more than a token amount of capacity to the screening task.

In a forensic context, where the analysis is performed on a memory snapshot, privacy is not really a concern and (unless a huge number of memory images are examined) even less efficient methods will do the job. However, we *do* have the additional concern of device/OS heterogeneity, which means that *genericity* (general applicability) is a

[*] Corresponding author.
E-mail addresses: vassil@roussev.net (V. Roussev), iahmed4@uno.edu (I. Ahmed), tsires@uno.edu (T. Sires).

http://dx.doi.org/10.1016/j.diin.2014.05.013
1742-2876/© 2014 Digital Forensics ResearchWorkshop. Published by Elsevier Ltd. All rights reserved.

highly desirable property. In particular, the ability to work with other architectures, *ARM* in particular, is becoming increasingly important as a great variety of devices use customized versions of the *Linux*/*ARM* platform.

As our further discussion will show, prior methods fail to satisfy all of the requirements, whereas the presented new approach does. In summary, the main contributions of this work to the field are three-fold:

- We survey a representative pool of Linux kernels and quantify their level of similarity and dissimilarity. The results point to gaps in the evaluations performed by prior research, and strongly suggest that prior techniques have not been evaluated at a significant level of granularity, and may actually fail to precisely distinguish among closely related kernels.
- Based on the results, we devise a new content-based technique and tool, called *sdkernel*, which produces a signature that is, on average, 0.3% of the size of the on-disk kernel image. In quick mode, the tool can screen a 1GiB image against 520 kernel signatures per second, on a single core.
- We present a detailed evaluation, which demonstrates that *sdkernel* produces robust results with *no false positives* and *no false negatives*, even for nearly identical kernel versions.

Related work

This section presents the existing OS fingerprinting techniques that are based on memory analysis and thus, close to sdkernel and, other techniques that generally use CPU, file system and other sources for kernel version identification.

Memory analysis-based fingerprinting techniques

Gu et al. (2012) propose *OS-Sommelier*, which is the most complete and sophisticated solution attempted to-date. The tool relies on detailed knowledge of the operation of the *Intel x86* in order to find enough variation in the basic implementation mechanisms to characterize individual kernels. This is accomplished by extensively parsing and analyzing the memory snapshot for kernel version identification such as virtual to physical address translation, and disassembling the code. The creation of the kernel's signature consists of three main steps:

The first step searches the entire snapshot and identifies the page global directory (PGD) for virtual to physical address translation. The second step identifies the kernel code – the PGD is used to make clusters of the similar read-only pages, assuming that the kernel-code pages are read-only for code protection. Further, the cluster is identified that contains core-kernel code, not kernel module code, since the same module can be loaded with different kernel versions. Two particular instructions are used that empirically identified as used only by the core kernel and not by the modules.

The third step generates the signatures, which are the cryptographic hashes of the kernel pages. Since the kernel code contains pointers whose values may change when the kernel loads in different locations in the memory. Thus this step also involves zeroed out such pointer values before computing hashes to neutralize the effect of the address space randomization; the code is disassembled to identify the locations of the pointers. In the end, a database of signatures is created representing different versions of code. Finally, when a kernel version needs to be identified in a memory dump, all the above steps are repeated on the dump to create signatures, which are then compared with the database to identify the kernel version.

Lin et al. (2011) propose *Siggraph* to identify kernel data structures from a memory dump. Since data structure definitions vary with different OSes, *Siggraph* can be used for OS fingerprinting. However, mostly well-known data structures do not change with minor version, which makes Siggraph only effective for detecting major kernel versions. Siggraph is also not efficient since it examines the data structure in many hierarchical levels and also analyze every field in a data structure.

Christodorescu et al. (2009) propose to use interrupt descriptor table (IDT) for kernel version identification. They analyze the table, which is an array of interrupt vectors containing pointers to interrupt handler code. Since interrupt handler code differ with different OSes, they compute the cryptographic hash values of handler code and use them as signatures to identify different OSes. Since their approach requires IDTR register value to directly identify the location of IDT, it does not work on memory dump that does not contain register values, i.e., it can only work on live systems.

Volatility is a popular framework for analyzing memory snapshots. It requires and maintains the profiles of kernel versions for parsing memory dump and applying right set of data structure definitions. In order to analyze a memory capture, it first needs to know the operating system version in the dump to select right profile for analysis. Volatility has the imageinfo tool to identify MS Windows versions such as Windows XP SP1, Windows XP SP2, Windows 7, etc. The tool scans the whole memory dump using predetermined signatures to find kernel debugging symbols table, which also contains the exact kernel version information. The *Volatility* approach is quite fragile in that the signature values can be altered to make *Volatility* not find the table. Moreover, the values in the table can also be modified to give *Volatility* an impression of a wrong kernel version.

Dolan-Gavitt et al. (2009) propose an automated method to build robust signatures of data structures. They identify the candidate fields, which are often accessed and used by OS. The fuzzing is then used to modify the candidate field values. If the modifications destabilize the OS or at least the functionality attached to the data structure, the field is then chosen for signature. However, the approach is not effective against the read-only data structures as such these are only used for debugging purposes.

Other fingerprinting techniques

Quynh proposes *UFO*, which is kernel code-independent OS fingerprinting technique and particularly uses CPU register states for kernel version identification. *UFO* utilizes the fact that the protected mode of Intel platform enforces no constraint on how OS is implemented. Thus, different OS versions have different implementations of setting up low-level data structures and other details such as global and interrupt descriptor tables (GDT and IDT) (Russinovich et al., 2009; Love, 2010), apparently making the values of registers, such as base and limit values of GDTR and IDTR, different. Furthermore, *UFO* uses fuzzy approach, where instead of matching exact signature, it gives weight to each parameter in a signature. While matching signatures, it computes the sum of the weight of each parameter matched and the signature with highest weight represents the kernel version.

Gu et al. (2012) evaluated *UFO* on different versions of Linux and Windows kernel and it appears that *UFO* does not work well on many Windows kernels and close version of Linux kernel. Moreover, *UFO* is restricted to work only on a live system and normally cannot be used on memory dumps.

There are number of OS fingerprinting tools (such as nmap and Xprobe2) that remotely identify kernel versions of a target system based on the packets being exchanged. For instance, *nmap* sends crafted packets with uncommon TCP options to target system. Since TCP/IP stack is implemented somewhat differently by different operating systems, the differences in the reply of the crafted packets are used to identify the OS versions. Similarly, *Xprobe2* uses ICMP packets for kernel version identification. These techniques however, are not effective to detect minor versions of kernel.

One most certain approach to identify the kernel version is through examining the file system on the hard disk of target system. For instance, one could maintain a database of hash values of different version of kernel code file, which then can be used to find the hash value of the kernel file in the hard disk of target system. The corresponding kernel version of matched hash value is the version of the kernel running on the target system. *Virt-inspector* (virt-inspector) is an exemplar tool that provides the capability to identify the kernel version on hard disk, USB, CD etc. It uses *libguestfs* (libguestfs) library to examine the file system on any non-volatile media.

Summary

The main challenge in building precise fingerprints is the use of *address space layout randomization* (ASLR) as a means of combatting buffer overflow attacks. Specifically, if the underlying system uses primarily *relocatable code* (e.g., *MS Windows*), this means that the memory-resident code is a modified version of the one on disk. The prior state-of-the-art solution, *OS-Somelier*, relies on a complicated – and potentially fragile – sequence of steps to compensate for ASLR during fingerprint generation. It is the result of substantial reverse engineering and low-level operational analysis; as such, it is highly specific to the *x86* architecture.

Brief similarity study of Linux kernels

Before we proceed with our design, we first perform a quick survey of Linux kernels in order to gain a basic understanding of what level of selectivity is possible, and what kind of expectations can reasonably be accommodated. We are interested in two base cases—*stock kernels* and *custom kernels*. The former represent (the common case of) pre-compiled kernels that are automatically installed by major *Linux* distributions' software management tools; the latter attempts to understand the effects of customization.

Table 1
Distribution of Ubuntu stock kernel version samples.

Kernel range	*Samples*	*Ubuntu version*
2.6.32-21–2.6.32-56	36	10.04
3.0.0-12–3.0.0-32	21	11.04
3.2.0-23–3.2.0-59	35	12.04
3.5.0-17–3.5.0-46	28	12.10
3.8.0-19–3.8.0-35	17	13.04
3.11.0-12–3.11.0-17	5	13.10
3.13.0-7–3.13.0-8	2	13.10

Stock kernels: Ubuntu x86

For our first study, we chose the *Ubuntu* collection of `linux-image` packages available online.[1] The rationale here is straighforward—*Ubuntu* is the most popular *Linux* distribution, its package numbering closely follows the mainline *Linux* kernel, and the packages are deployed in an automated fashion. In other words, it is a representative case of a *stock kernel*.

Out of the 943 available packages, we selected all the *generic* versions—a total of 300 packages (150 32-bit and 150 64-bit kernels with matching versions). After unpacking them, we ended up with 288 unique kernel images (144 for each architecture).

Table 1 provides a basic breakdown of the data set. The first column provides the range of version numbers, the second gives the number of samples in the range (a few versions are not present), and the third gives the default *Ubuntu* version associated with the kernel. The set encompasses both *long term support* (LTS) releases—10.04, 12.04—which still receive updates, as well as regular 6-month releases, which are phased out quickly.

We should not that these packages are not necessarily a perfect historical record—some of them contain backported changes (bug fixes) and some of the minor release versions are not part of the record at all. Also, it is likely that different compiler versions have been used at different times.

Similarity & uniqueness

Since a *vmlinux* (kernel) image is loaded into a *tmpfs* instance, it is meaningful to examine its contents in page-sized (4KiB) chunks. For that purpose, we produce the 4K-block-aligned crypto hashes for each kernel version and then calculate the number of unique (across the entire set) blocks for each kernel.

[1] http://security.ubuntu.com/ubuntu/pool/main/l/linux/.

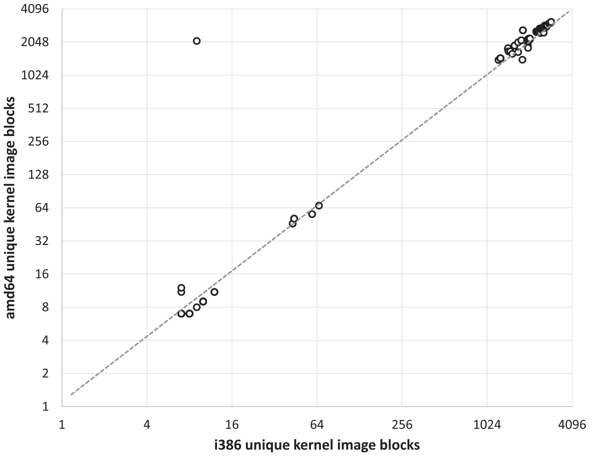

Fig. 1. Scatter plot of unique blocks per kernel: `i386` vs. `amd64`.

Fig. 1 correlates the number of unique blocks in the 32-bit (`i386`) and 64-bit (`amd64`) builds of each version. The picture yields several interesting observations:

- With one exception, the number of unique blocks for a given version correlates almost perfectly across the two architectures. In retrospect, this is logical as the images are built from the same source and use the same tool chain. Therefore, we conclude that picking one of the architectures would be sufficient to represented the characteristics of both.

We do not have a conclusive explanation for the large difference in unique blocks between the `i386` and `amd64` versions of the 2.6.0-52 kernel. There are a number of possible explanations with differences in the build environment as the most likely culprit. In any case, we only use the results to reduce the data presented in the paper; the observed deviation has no real bearing on the effectiveness of the presented method.

- The vast majority, 112 out 144 of the kernels have a large number (at least 1230) of unique blocks. In other words, the changes introduced from version to version are large enough to produce between 36 and 66% unique blocks.
- The remaining 32 kernel versions have *very few*, as low as seven, unique blocks. This strongly suggests that changes to the code between some versions are quite minimal, making them an inherently difficult classification problem.

Table 2
Kernel images with low number of unique blocks.

Kernel	i386	amd64	Kernel	i386	amd64
2.6.32-47	7	11	3.5.0-32	7	7
2.6.32-48	7	12	3.5.0-33	7	7
2.6.32-53	59	56	3.5.0-34	7	7
2.6.32-54	66	67	3.5.0-38	43	46
3.2.0-44	8	7	3.5.0-39	43	46
3.2.0-45	8	7	3.5.0-41	7	7
3.2.0-47	8	7	3.5.0-42	7	7
3.2.0-48	8	7	3.8.0-19	12	11
3.2.0-49	8	7	3.8.0-20	10	9
3.2.0-50	7	7	3.8.0-21	12	11
3.2.0-51	7	7	3.8.0-22	10	9
3.2.0-52	9	2095	3.8.0-23	10	9
3.5.0-28	10	9	3.8.0-24	10	9
3.5.0-29	9	8	3.8.0-25	10	9
3.5.0-30	10	9	3.8.0-28	44	51
3.5.0-31	7	7	3.8.0-29	44	51

Table 3
Kernels used in *arm5* experiments.

Version	Count	Version	Count	Version	Count
3.00	96	3.05	8	3.10	30
3.01	11	3.06	12	3.11	11
3.02	55	3.07	11	3.12	11
3.03	9	3.08	14		
3.04	80	3.09	12		

To provide further context for the latter observation, Table 2 lists the specific version for all points where at least one of the architectures has fewer than 100 unique blocks. The clustering data shows that, indeed, some groups of consecutive stock kernels are nearly identical.

Custom kernels: arm5

For our second study we cross-compiled 357 Linux kernel versions for the *arm5* architecture in two configurations each: *default* and *qemu*. The former uses all the default build options, whereas the latter uses a configuration necessary to run the kernels in QEMU.[2] Table 3 provides a histogram of the kernels used, binned by major version number.

The two kernel sets allow us to ask a basic question—how similar/distinct are kernels compiled from the same source but with different options (something a custom kernel developer might do)?

As it turns out, the configuration does have a major impact on our similarity measure and the effect is remarkably consistent across the entire set. To measure it, we similarity-compared the corresponding *qemu* and *default* build of each kernel version.

Table 4 provides a summary of the results grouped by major version. For example, to obtain the first row of numbers describing the 3.00 kernel version, we considered the series

$$sdhash\left(K^{3.00.01}_{qemu}, K^{3.00.01}_{default}\right), \ldots, sdhash\left(K^{3.00.12}_{qemu}, K^{3.00.12}_{default}\right),$$

where K the kernel with the given configuration name and version number.

The mean score for all series is in the 12 to 13 range, with very tight deviation bounds. Such `sdhash` scores are a very weak indicator (essentially noise) thereby implying that differently configured kernel builds can easily diverge substantially in their outcome. (We observed a similar phenomenon with *x86* kernels – the *generic* version of the kernel tended to be more similar to neighboring *generic*

[2] http://qemu.org.

Table 4
Kernel sdhash score statistics: qemu vs. default configuration.

Version	Mean	StDev	Version	Mean	StDev
3.00	12.46	1.06	3.07	13.00	0.60
3.01	12.82	1.03	3.08	12.43	1.12
3.02	12.40	0.84	3.09	13.00	1.00
3.03	12.83	0.69	3.10	12.83	1.00
3.04	12.65	0.71	3.11	12.64	1.15
3.05	12.38	0.87	3.12	13.18	1.64
3.06	12.75	0.60			

versions than to other variations, such as *lowlatency*, of the same kernel number.)

Summary

It appears that prior research has not considered kernel versions at this level of granularity. Such consideration is important for at least two reasons:

a) It shows that the evaluation of prior methods is likely incomplete; indeed, it is quite possible that they would fail to work at this level of granularity, despite claims to perfection (e.g., Gu et al., 2012).
b) Custom kernels can produce potentially very different outcomes based solely on build options. This is both good and bad news—it suggests that it is feasible to build very specific kernel signatures, but also implies that it might be difficult to relate a custom kernel for which we have no base to a known standard one.

Building content-based signatures

Similarity digest comparison

The concept of similarity digest (Roussev, 2010) and its implementation `sdhash` were developed to provide *bytewise approximate matching* (Breitinger et al.) of arbitrary data objects. Specifically, they are designed to support two kinds of queries—*resemblance* and *containment*. In the former, the objects compared are of similar size and the query effectively seeks to estimate their level of commonality. In the latter scenario, the objects compared are very different in size and the purpose is to establish wether the small object is contained in the larger one.

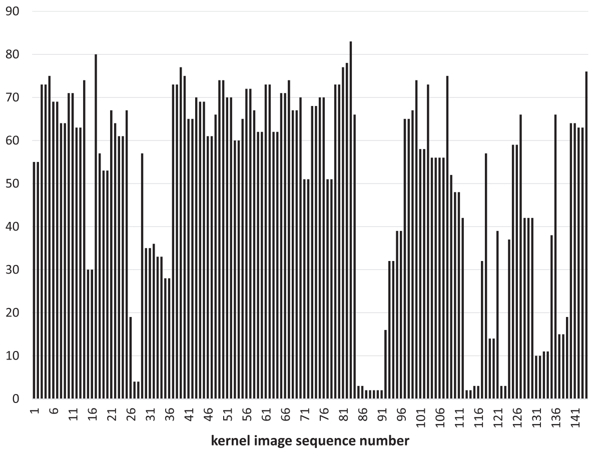

Fig. 2. Selectivity of `amd64` signature bases.

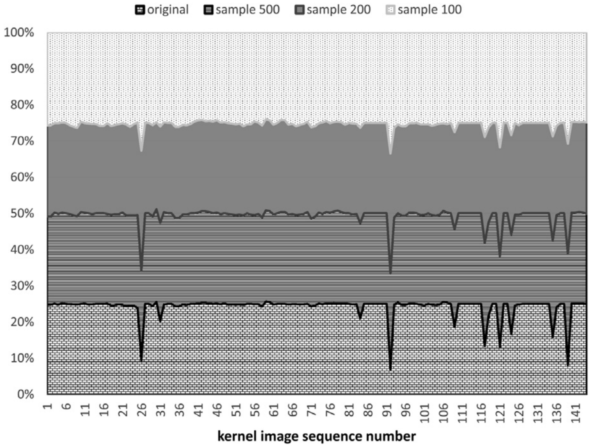

Fig. 3. Relative selectivity of original and *sampled* `amd64` signature bases.

Clearly, kernel identification can be modeled as a containment query where we look for the presense of known kernel content in a RAM snapshot. Identifying known files in RAM has already been demonstrated for triage purposes (Roussev and Quates, 2012) so the basic rationale for using `sdhash` in this scenario is not a big stretch. Yet, there are a couple of practical considerations that require additional work.

Kernel similarity. As already discussed, OS kernels minor releases for Linux systems can be quite similar to each other, thereby requiring some preprocessing to build a distinct fingerprint.

Throughput. Among the data set we are working with, 64-bit kernels can reach 20 MB and a realistic RAM snapshot can be several GB. The naive application of `sdhash` would not be particularly performant. For example, comparing 16 32-bit Windows kernels (55 MB) against 9 RAM snapshots (18 GB) on 4 cores (3 GHz Intel X5670) takes 18 m 23 s.

Fortunately, both of these concerns point in the same direction—we need to minimize the size of the signature, while retaining a useful level of selectivity.

Building a kernel signature

The rationale behind our signature-building process is twofold—eliminate repetitive content across kernels and downsample the results to the smallest size that retains robust selectivity. The specific procedure we use consists of the following steps:

1. *Obtain a set of kernel images that are of interest.* These should include as many variations as possible in order to facilitate the creation of the most specific fingerprints.
2. *Split the images into page-sized blocks and eliminate all non-unique blocks.* This step has also the benefit of eliminating all blocks filled with zeroes and other

Table 5
Selectivity deviations due to sampling.

Kernel #	Original	s-500	s-200	s-100
26	19	50	67	66
31	35	47	46	45
84	66	83	83	83
92	16	62	77	78
109	52	75	75	77
117	32	68	70	69
118	57	68	69	69
121	39	74	89	94
124	37	61	61	63
135	38	64	69	69
139	19	72	71	72

common blocks with low information content. As our later discussion will show, this step is critical in building distinct signatures for closely related kernels in the set.

3. *Create a sample of the blocks of the desired size from each image.* The size of the sample could be chosen based on absolute size (e.g., 100 4K blocks), or determined as a fraction of the size of the image (e.g., 5%). In the evaluation section, we explore the trade off between sample size and precision.
4. *Create a block-based* `sdhash` *similarity digest.* We study and recommend parameters that optimize size, throughput, and selectivity.

The first two steps are performed in the same manner as in the empirical experiments on Section 3. Therefore, we focus our discussion on the last two steps in the process.

Selectivity & sample size

For kernels with low number of unique blocks (7–67, in our *Linux x86* sample), it should be clear that the appropriate approach is to include all the available blocks in order to maximize our ability to discern closely related kernels.

To understand how much data we should retain from the rest (with 1200 + unique blocks), we performed a cross-similarity study at various levels of sampling using our Linux data set. We start by obtaining a *signature base*, which consists of (the concatenation of) all unique blocks for each image. We cross-compare the resulting files using (4KiB) block-aligned `sdhash`, and compute the following measure of selectivity:

Table 6
MS Windows versions used in the evaluation.

Short name	Version description
`xp.2`	WindowsXP, Service Pack 2
`xp.3`	WindowsXP, Service Pack 3
`vista.0`	Windows Vista
`vista.1`	Windows Vista, Service Pack 1
`vista.2`	Windows Vista, Service Pack 2
`win7.0`	Windows 7
`win7.1`	Windows 7, Service Pack 1
`win8.0`	Windows 8
`win8.1`	Windows 8.1

Table 7
Kernel versions used in the evaluation.

OS version	Kernel file	Short kernel name
`xp.2`	`ntoskrnl.exe`	`xp.2-os`
`xp.2`	`ntkrnlpa.exe`	`xp.2-pa`
`xp.3`	`ntoskrnl.exe`	`xp.3-os`
`xp.3`	`ntkrnlpa.exe`	`xp.3-pa`
`vista.0`	`ntoskrnl.exe`	`vista.0-os`
`vista.0`	`ntkrnlpa.exe`	`vista.0-pa`
`vista.1`	`ntoskrnl.exe`	`vista.1-os`
`vista.1`	`ntkrnlpa.exe`	`vista.1-pa`
`vista.2`	`ntoskrnl.exe`	`vista.2-os`
`vista.2`	`ntkrnlpa.exe`	`vista.2-pa`
`win7.0`	`ntoskrnl.exe`	`win7.0-os`
`win7.0`	`ntkrnlpa.exe`	`win7.0-pa`
`win7.1`	`ntoskrnl.exe`	`win7.1-os`
`win7.1`	`ntkrnlpa.exe`	`win7.1-pa`
`win8.0`	`ntoskrnl.exe`	`win8.0-os`
`win8.1`	`ntoskrnl.exe`	`win8.1-os`

Let $b_i, i = 1..n$ be the signature bases for all n images under consideration, and $sdhash_{4k}(b_i, b_j), i,j = 1..n$ be the `sdhash` similarity score between b_i and b_j when compared with 4K-blocked-aligned version of the algorithm. (Note: $sdhash(x,y) = sdhash(y,x)$, by design.)

We define the *selectivity* $sel(b_i)$ of a signature base b_i as

$$sel(b_i) = 100 - \max\left(sdhash_{4k}\left(b_i, b_j\right) : 1 \le j \le n, j \ne i\right).$$

In other words, we use the minimum difference between a self-similarity test ($sdhash_{4k}(b_i, b_i) = 100$) and any other similarity comparison as a measure of selectivity; the higher the difference, the greater the selectivity.

Fig. 2 shows the baseline level of selectivity for all 144 `amd64` kernels (`i386` results look nearly identical). The most important observation is that in *all* cases, the selectivity is positive, implying that the kernels *can* be successfully identified using the chosen signature base. For some of the kernels, however, the selectivity is quite small; unsurprisingly, these clusters correspond to the kernel versions with very low numbers of unique blocks cited earlier (Table 2).

Having convinced ourselves that signature bases provide a sound foundation for building a fingerprint, our next task is to explore the feasibility of shrinking the fingerprint without incurring a precision penalty.

Let $s(k,b)$ be a sampling function, which—given sample size k and signature base b—returns a *sampled base* consisting of k uniformly chosen blocks from b. If b has fewer than k blocks, or $k = 0$, the original data is returned.

To evaluate the effect of sampling on selectivity, we compute $sel(s(k,i))$, where $k = 0, 100, 200, 500$ and $i = 1...n$. In other words, we compare the selectivity of the entire signature base to that of sampled bases consisting of 100, 200, and 500 blocks, respectively. Fig. 3 illustrates the results in the form of a stacked chart, which compares in relative terms the comparison results for each kernel.

In the ideal case, sampling would have no effect on the scores and the chart would look like four identical horizontal strips, each occupying 25% of the chart. We can see that about 92% of the time (133 out of 144 cases) the four areas are very close to equal, which implies that we can

Table 8
Similarity scores for the full MS Windows kernels. Numbers in bold are the highest score per row.

	xp.2	xp.3	vista.0	vista.1	vista.2	win7.0	win7.1	win8.0	win8.1
xp.2-os	15	15	13	14	14	13	13	13	13
xp.2-pa	**44**	21	13	14	14	13	13	13	13
xp.3-os	15	17	13	14	13	13	13	13	13
xp.3-pa	21	**45**	14	14	14	13	13	13	13
vista.0-os	12	12	18	17	17	14	14	13	13
vista.0-pa	12	12	**35**	17	17	14	14	13	13
vista.1-os	12	12	16	19	18	15	15	13	13
vista.1-pa	12	12	16	**61**	18	15	15	13	13
vista.2-os	11	12	16	19	19	15	15	13	13
vista.2-pa	12	12	16	20	**37**	14	15	13	13
win7.0-os	11	12	14	15	15	17	18	13	13
win7.0-pa	11	12	14	15	15	**38**	18	13	13
win7.1-os	11	12	14	15	14	18	22	13	13
win7.1-pa	11	12	14	15	15	17	**31**	13	13
win8.0-os	11	11	12	13	13	13	13	**34**	15
win8.1-os	11	11	12	13	13	13	13	15	**53**

Table 9
Similarity scores for sampled MS Windows kernels: s-100.

	xp.2	xp.3	vista.0	vista.1	vista.2	win7.0	win7.1	win8.0	win8.1
xp.2-os	13	14	13	14	14	13	13	13	13
xp.2-pa	**44**	21	14	14	14	14	13	13	13
xp.3-os	16	17	13	14	13	13	13	13	13
xp.3-pa	22	**46**	14	15	14	13	13	13	13
vista.0-os	11	12	17	16	15	13	13	13	12
vista.0-pa	12	12	**36**	19	19	14	14	13	13
vista.1-os	12	12	16	19	17	15	15	13	13
vista.1-pa	12	12	16	**64**	19	16	16	13	14
vista.2-os	12	12	17	20	19	16	16	14	14
vista.2-pa	11	12	16	19	**37**	13	13	13	13
win7.0-os	12	12	15	16	15	17	18	13	14
win7.0-pa	11	12	14	15	14	**40**	18	13	13
win7.1-os	11	12	14	15	14	17	20	13	13
win7.1-pa	11	11	13	14	14	18	**32**	13	13
win8.0-os	11	11	12	13	12	12	12	**34**	14
win8.1-os	10	11	12	13	13	13	13	16	**55**

downsample the base all the way to 100 blocks with no adverse effects on precision. This is a substantial result as it reduces the size of signature (and the time to compare it) 12–32 times for our reference set.

In the rest of the cases, we can observe some non-trivial deviations from the ideal case and Table 5 provides the details. The first column gives the kernel sequence number (consistent with Fig. 3), the second column provides the selectivity in the unsampled case, whereas the columns labeled *s-500*, *s-200*, and *s-100* provide the selectivity score for 500/200/100-block samples, respectively. We can see that the deviations can be substantial; however, the good news is that the selectivity actually goes up (i.e., differences are exaggerated) so it does not negatively affect the outcome.

Table 10
Similarity scores and selectivity for base case and for 25x16KiB samples.

OS version	$sdhash_{base}$	sel_{base}	$sdhash_{16k}$	sel_{16k}
xp.2	44	23	31	17
xp.3	45	24	23	9
vista.0	35	17	37	19
vista.1	61	42	31	14
vista.2	37	17	29	12
win7.0	38	20	30	18
win7.1	31	9	38	21
win8.0	34	19	21	6
win8.1	53	38	27	17

Evaluation

In this section, we use three different platforms to comprehensively evaluate the proposed solution with respect to precision and throughput.

MS Windows (x86)

For the first set of experiments we used nine different 32-bit versions/service packs of the MS Windows (x86 architecture) and 16 different kernels that were available on the systems. Tables 6 and 7 provide a details list if these,

Table 11
amd64: Similarity scores and selectivity for base case and for 25x16KiB samples.

OS version	$sdhash_{base}$	sel_{base}	$sdhash_{16k}$	sel_{16k}
2.6.32-35	67	33	61	32
2.6.32-42	67	26	61	26
2.6.32-48	67	18	61	9
2.6.32-54	67	0	61	0
3.2.0-31	68	23	62	24
3.2.0-38	68	41	62	40
3.2.0-45	68	0	62	0
3.2.0-53	68	17	62	18

and introduce shorthand notation used in the result presentation.

Tables 8 and 9 present the results of comparing the each of the 16 kernels on disk (rows) against nine 2 GB RAM snapshots (columns). Most of the distributions come with two different versions of the kernel of which only one would be active and identifiable in the snapshot. Note that the score along the diagonal would not be 100 because the RAM layout of the image is different from the on-disk serialization.

Intuitively, Windows kernels should provide an easy target—they change (officially) relatively infrequently so the expected differences should be easy to spot. Indeed, Tables 8 and 9 illustrate the point well. After downsampling the kernels to approximately 100 (unique) 4KiB blocks—an almost eightfold reduction from 55 MB to 7 MB)—the similarity scores (and selectivity) remain almost identical. The compute time (on four cores) is correspondingly reduced 8.6 times (from 1103 to 128 s).

To further reduce compute time, we increase the block size at which the `sdhash` score is computed from 4KiB to 16KiB; this has the effect of packing four times as many features into the same signature (and is within the design parameters of the tool (Roussev, 2012)). We use a sample of 25 blocks of 16KiB each to construct the kernel signatures.

Table 10 shows the results; to conserve space we give the highest score (which in *all* cases comes from the correct pairing) and the selectivity measure (the difference to the second highest) for both the baseline and the 16KiB-sample case. We observe that, while the absolute similarity change, the level of selectivity remains more than adequate to easily distinguish the best match.

The execution time, however, drops from 128 to 9.7 s on four cores. Using the time for single-core execution (26.2 s), we can calculate that our approach needs 1.44 s per GB to screen a RAM snapshot for the 16 Windows kernels.

Linux (x86)

For this evaluation we picked a sampling of approximately equally-spaced kernel releases from the 2.6.32 and 3.2.0 series. In particular, we included the 2.6.32-54 and 3.2.0-45 kernels, which we expect to be difficult cases based on the results in Table 2. Indeed, we can see from the results in Table 11 that, in the average case, the digests behave quite well but in the two extreme cases, we are unable to distinguish the almost identical kernels (recall that they have seven unique 4K blocks). Sampling at 16KiB does not introduce any notable selectivity degradation.

Linux (arm)

Before we proceed the accuracy of the kernel identification in a RAM snapshot, we perform a basic uniqueness check on the 97 3.00 kernels. After block deduplication, there are 27 kernels that have only five, or six unique 4K blocks; this suggests that disambiguating them in a RAM snapshot is an inherently difficult proposition. We self-compared the 3.00 kernel set to determine if there are any false results.

We find that there are no false matches at the 100 similarity level in both the base and 16K-sampled cases. However, there are 18 pairs that yield a score of 90, or above (most are 99), which suggests that they would be potentially difficult targets.

To keep the effort reasonable, we limit our evaluation to RAM snapshots of the 3.12 (*qemu*) kernel series, which is representative of recent trends in the number and frequency of minor version per major one. Table 12 shows base and aggressively (16KiB) sampled scores and selectivity measures. In all scenarios, no false positive/negatives are introduced, while computation is sped up approximately 100 times.

Discussion

One aspect that has not been explored in the above evaluation is changes to the tool chain, such as changing the compiler (or its code generation options). Major changes to the build environment (e.g., optimization level) would result in different code output. Consequently, it would not be possible to correlate these different versions (and we are not aware of any work that would be able bridge that gap). In other words, the notion of *kernel version* in this context should be augmented to include configuration/build options, and the specific tool chain used. Also, it should be clear that `sdkernel`, like prior work, can only identify kernel versions it knows about (or very similar ones).

Table 12
arm5: Similarity scores and selectivity for base case and for 25x16KiB samples.

OS version	$sdhash_{base}$	sel_{base}	$sdhash_{16k}$	sel_{16k}
3.12.00	82	22	77	22
3.12.01	81	21	77	20
3.12.02	68	21	65	29
3.12.03	71	24	70	35
3.12.04	87	14	85	17
3.12.05	87	15	85	17
3.12.06	71	26	72	30
3.12.07	71	19	69	26
3.12.08	75	25	68	24
3.12.09	77	36	70	37
3.12.10	71	22	70	37

Conclusion

In this paper we introduced a content-based method for reliably identifying kernel versions. Our work makes several contributions to the field:

• We performed a similarity survey of several hundred Linux kernels for both the *Intel x86* and *ARM* architectures. We showed that, while most kernel versions are reasonably different, a fraction of the releases are nearly identical—-with less than 40KiB of unique content; such releases are inherently difficult to discern from each other in a RAM snapshot.

Relatively modest changes to the build setup can result in very distinct kernel versions (from a data content point of view).

Evaluations of prior work do not include sufficient level of granularity in order to make a claim to perfection.

- Based on our observations, we developed a practical approach to building and optimizing kernel signatures by utilizing similarity digests. The signature is derived from the on-disk representation of the kernel and can be used to screen RAM snapshots. The developed approach leverages existing tools and can be deployed immediately.
- The technique works reliably and, with the exception of near-identical kernels, produces no false positive or false negative results. In the case of near-identical kernels, the algorithm degrades gracefully in that it produces (nearly) identical scores.
- The method affords excellent throughput—it can screen 1 GB/s of RAM against 11 kernel signatures on a single core. Given more processing power, the computation can be scaled up as `sdhash` has a parallelized implementation.
- Unlike prior work, which relies heavily on reverse engineering and detailed understanding of how the underlying hardware and OS operate, the developed approach is architecture-agnostic. It does not require any reverse engineering efforts to work; by extension, it does not require maintenance as kernels evolve over time.

References

F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, Approximate matching: definition and terminology. URL http://csrc.nist.gov/publications/drafts/800-168/sp800_168_draft.pdf.

Christodorescu M, Sailer R, Schales DL, Sgandurra D, Zamboni D. Cloud security is not (just) virtualization security: a short paper. In: Proceedings of the 2009 ACM workshop on Cloud Computing Security, CCSW'09. New York, NY, USA: ACM; 2009. pp. 97–102. http://doi.acm.org/10.1145/1655008.1655022.

Dolan-Gavitt B, Srivastava A, Traynor P, Giffin J. Robust signatures for kernel data structures. In: Proceedings of the 16th ACM conference on Computer and Communications Security. ACM; 2009. pp. 566–77.

Gu Y, Fu Y, Prakash A, Lin Z, Yin H. Os-sommelier: memory-only operating system fingerprinting in the cloud, in. In: Proceedings of the third ACM symposium on Cloud Computing, SoCC'12, ACM, New York, NY, USA. pp. 5:1–5:13, http://doi.acm.org/10.1145/2391229.2391234; 2012.

imageinfo, https://code.google.com/p/volatility/wiki/CommandReference#imageinfo.

libguestfs, http://libguestfs.org/.

Lin Z, Rhee J, Zhang X, Xu D, Jiang X. Siggraph: brute force scanning of kernel data structure instances using graph-based signatures. NDSS. http://dx.doi.org/10.1.1.188.7659; 2011.

Love R. Linux kernel development. 3rd ed. Addison-Wesley Professional; 2010.

nmap, http://nmap.org/.

N. A. Quynh. [link]. URL http://www.defcon.org/images/defcon-18/dc-18-presentations/Quynh/DEFCON-18-Quynh-OS-Fingerprinting-VM.pdf.

Roussev V, Quates C. Content triage with similarity digests: the m57 case study. In: Proceedings of the 12th Annual Digital Forensics Research Conference. S60–8. http://dx.doi.org/10.1016/j.diin.2012.05.012; 2012.

Roussev V. Data fingerprinting with similarity digests. In: Advances in digital forensics VI. Springer; 2010. pp. 207–26.

Roussev V. Managing terabyte-scale investigations with similarity digests. In: Advances in digital forensics VIII. Springer; 2012. pp. 19–34.

Russinovich ME, Solomon DA, Ionescu A. Windows internals: including Windows server 2008 and Windows Vista. 5th ed. Microsoft Press; 2009.

virt-inspector, http://libguestfs.org/virt-inspector.1.html.

Volatility, https://code.google.com/p/volatility/.

Xprobe2, http://sourceforge.net/projects/xprobe/files/xprobe2/.