Robust Fingerprinting for Relocatable Code

Irfan Ahmed, Vassil Roussev, Aisha Ali Gombe
Department of Computer Science
University of New Orleans
2000 Lakeshore Dr.
New Orleans LA USA - 70148
irfan.ahmed@uno.edu, vassil@cs.uno.edu, aaligomb@my.uno.edu

ABSTRACT

Robust fingerprinting of executable code contained in a memory image is a prerequisite for a large number of security and forensic applications, especially in a cloud environment. Prior state of the art has focused specifically on identifying kernel versions by means of complex differential analysis of several aspects of the kernel code implementation.

In this work, we present a novel technique that can identify *any* relocatable code, including the kernel, based on inherent patterns present in relocation tables. We show that such patterns are very distinct and can be used to accurately and efficiently identify known executables in a memory snapshot, including remnants of prior executions. We develop a research prototype, `codeid`, and evaluate its efficacy on more than 50,000 sample executables containing kernels, kernel modules, applications, dynamic link libraries, and malware. The empirical results show that our method achieves almost 100% accuracy with zero false negatives.

Categories and Subject Descriptors

K.6 [**Management Of Computing And Information Systems**]: Security and Protection; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software, Security kernels, Verification*

Keywords

code fingerprinting; codeid; memory analysis; virtual machine introspection; cloud security; malware detection

1. INTRODUCTION

Identifying executable binary code–operating system (OS) kernels, libraries, applications, malicious code (*malware*)–is the main focus of almost all (proactive) security monitoring, and (reactive) forensic analysis applications. For instance, fingerprinting of the OS kernel enables memory forensic tools (such as Volatility [18, 5]) to parse the memory with the data structures pertinent to the kernel version; fingerprinting of malware is at the core antivirus applications. Most incident response and deep forensics techniques rely heavily on the ability to recognize the exact code being executed on the target system.

In an *Infrastricture-as-a-service* (IaaS) *cloud* environment, (executable) code fingerprinting is a critical tool that enables a large number of automated services, such as patch management and host intrusion detection/prevention. More generally, the trend is towards cloud providers providing ever more sophisticated security-related services for their tenants. Indeed, providers who monitor a large population of virtual machines (VMs) are in a much better position to track attacks in progress, and proactively identify and shield other vulnerable VMs. This applies not only to network services, but also to client code in virtualized workstations.

All such services are reliant on their ability to gather an inventory of running code inside the VMs. Although it is possible to obtain some of this information indirectly by network scanning, or by examining the file system, such approaches are inherently incomplete as most applications do not provide a network service, and tenants routinely encrypt file system content. Thus, the only reliable approach is to directly examine the physical memory of the VM.

In this paper, we present *CodeIdentifier* (`codeid`)–a new fingerprinting technique and tool for relocatable code that identifies memory-resident code based on the content of relocation tables. *CodeIdentifier* divides an executable file into memory-page size blocks, and generates signatures for the blocks. A block signature consists of two components: 1) relocations (or pointers) in the code of the block identified by relocation table, and 2) the offset values of the relocations from the start of the block. The signatures are then used to identify the exact in-memory pages represented by the signatures. We have exhaustively evaluated `codeid` signatures on more than 50,000 sample executables (of kernel, kernel modules, applications, libraries, and malware), and established that the signatures are quite unique and are able to distinguish close versions of the code. The empirical results show that `codeid` detects in-memory code pages with almost 100% accuracy, and no false negatives.

The *main contribution* of this work is the development of `codeid`, which is a) *accurate*–it yields precise results, even for closely related code; b) *robust*–it works with partial information and can perform remnant detection; c) *performant*–the resulting tool is fast enough to be of practical use in scanning live VMs; and d) *fully automated*–it requires no human in the loop to generate and use new signatures.

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.
Copyright 2015 ACM 978-1-4503-3191-3/15/03...$15.00
http://dx.doi.org/10.1145/2699026.2699104.

The rest of the discussion is organized as follows: Section 2 provides a brief summary of related work; Section 3 presents the main concepts and techniques used in our solutions; Section 4 contains the experimental validation of the proposed approach; and Section 5 summarizes our findings and conclusions.

2. RELATED WORK

Prior work on executable fingerprinting techniques–apart from malware detection, which is outside the scope of this discussion–focus primarily on operating system (OS) kernel version detection.

2.1 Deep analysis techniques

Deep analysis approaches parse and (partially) interpret the memory capture with the express goal of finding unique implementation traits that separate different kernels.

Gu *et al.* [3] propose *OS-Sommelier*, which extensively parses and analyses memory dump for kernel version identification, such as virtual-to-physical address translation, and disassembling the code. In order to create a signature of a kernel, the tool takes a snapshot of the memory that contains the kernel, and then processes the snapshot in three main steps.

First, it searches the entire snapshot and identifies the page global directory (PGD) for virtual-to-physical address translation. Second, it 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. Next, the *cluster* which contains core-kernel code is identified; this is necessary since the same module can be loaded with different kernel versions. The authors use two particular instructions that are empirically identified as being used only by the core kernel, and not by modules.

Third, the tool generates the signatures, which are the cryptographic hashes of the kernel pages. Since the kernel code contains pointers whose values change when the kernel loads in different locations in the memory, this step is preceeded by zeroing out of all such pointer values before computing the hashes. To identify the locations of the pointers, the code is (partially) disassembled. Thus, a kernel signature consists of the obtained set of page hashes. Finally, when a kernel version needs to be identified in a memory image, the same steps are repeated on the target to create signatures, which are then compared with the known signatures. We discuss *OS-Sommelier*'s performance in more detail in our evaluation section.

Lin *et al.* [8] propose *Siggraph*, which relies on identifying in-memory kernel data structures. Since data structure definitions vary across operating systems, *Siggraph* can be used for OS fingerprinting. The main limitation is that well-known data structures tend to be stable across at least minor version releases. More generally, data structures changes are unpredictable and require manual reverse engineering effort whenever a new version comes out.

Christodorescu *et al.* [2] employ the *interrupt descriptor table* (IDT) for kernel version identification. The table is an array of interrupt vectors containing pointers to interrupt handler code. Since interrupt handler code tends to change over time, the authors compute the cryptographic hash of handler code, and use them as signatures to identify different kernels. Since this approach requires the value of the *IDTR* register to locate the IDT, it can only work on a live system.

Volatility [18] is a popular framework for deep forensic analysis of memory images. Since knowing the precise kernel version is critical for a lot of its functions, it maintains the profiles of kernel version for parsing memory captures and applying the correct set of data structure definitions. Volatility provides the `imageinfo` [4] tool to identify the various *MS Windows* versions. 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 has two main drawbacks: a) it is fragile in that the signature values can be altered to make Volatility not to find the table; and b) it is not fully automated and requires an expert analyst to drive the discovery process.

2.2 Other fingerprinting techniques

Quynh [13] proposes UFO–a kernel code-independent OS fingerprinting technique, which uses CPU register states for kernel version identification. UFO utilizes the fact that the protected mode of the Intel platform enforces few constraints on how OS is implemented. Thus, different OS versions use different ways of setting up low-level data structures and other details, such as global and interrupt descriptor tables (GDT and IDT) [15] [9] differently. As a results, the values of some registers, such as base and limit values of GDTR and IDTR, are different. UFO uses a fuzzy matching approach and looks for the closest match, giving different weights to each parameter in a signature. While matching signatures, it computes the sum of the weights of each parameter matched and the signature with highest weight represents the kernel version. In [3], Gu *et al.* evaluated UFO on different versions of Linux and Windows kernel and concluded that it does not work well on many *MS Windows* kernels and close versions of Linux kernel. Importantly, UFO is restricted to work only on a live system and cannot be used on memory captures.

Historically, there are a number of network fingerprinting tools (such as *nmap* [10] and *Xprobe2* [20]) that remotely identify the kernel version of a target system based on the packets being exchanged. Since the TCP/IP stack is implemented differently by operating systems, the differences in the reply of the crafted packets can be used to identify the OS versions. Such an approach is not relevant to our scenario and is, generally, inherently unreliable.

One accurate approach to identify the kernel version is to examine the file system on the hard disk of target system and, for instance, maintain a database of the file hashes for different version of the kernel. *Virt-inspector* [17] is a tool that provides the capability to identify the kernel version on hard disk, USB, CD etc. It uses libguestfs [6] library to examine the file system on any non-volatile media. The main constraint here is the requirement to access non-volatile media, which in our case is not given.

3. RELOCATION-BASED CODE FINGERPRINTING

3.1 Problem statement & requirements

Given a physical memory capture, our goal is to identify the presence of a known piece of code in it, which is running at an arbitrary location. Indeed, modern operating systems routinely use *address space layout randomization* (ASLR) and purposefully choose a random starting address. This

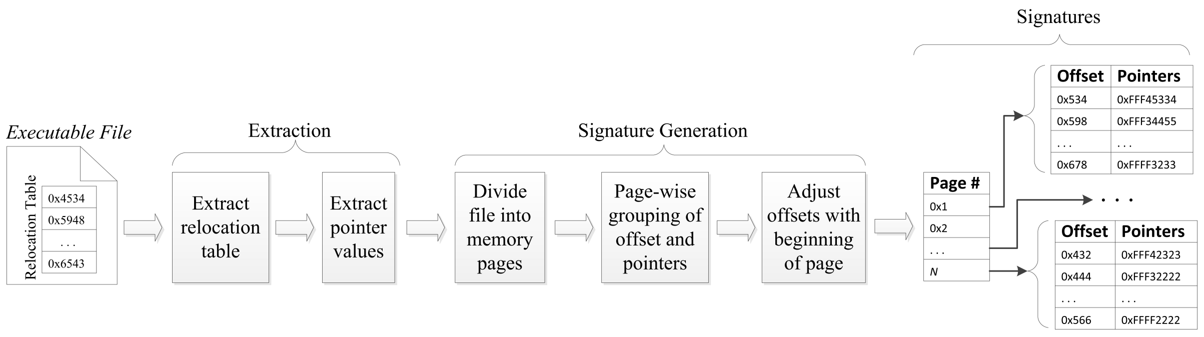


Figure 1: *CodeIdentifier* signature generation process

means that code must be able to execute regardless of its starting address. This property can be accomplished in one of two principle ways: a) by generating position-independent code–resulting in a *position-independent executable* (PIE)–that can be loaded at any address with no modifications to the code; b) by generating *relocatable code*–resulting in a *portable executable* (PE)–whose absolute addresses are adjusted at load time using the chosen base address value.

For PIEs, since no tranformations are applied to the code at load time, there is a direct mapping between the contents of the file containing the binary and the corresponding contents of RAM pages during execution. This leads to a straighforward approach to identifying the executable in memory: first, generate the (cryptographic) hash of all code pages for the binary, then compare the set to the hashes of all RAM pages of the target.

Therefore, in this work, we assume that the target code is in PE format, which is the case for nearly all kernel modules, libraries, and applications for the *MS Windows* platform.

Given our motivating problem of supporting security and management services in a large-scale virtualized environment, any practical solution should address at least three basic requirements:

Accuracy. The solution must have very low false positive/negative rates, and must be sensitive enough to distinguish among different version releases of the same executable.

Robustness. The solution should be able to work with partial information in order to accommodate paging effects and remnant detection from completed executions.

Throughput. The solution must have high throughput and low overhead to facilitate fast response and low implementation cost.

3.2 Overview

Recall that relocation is the process of assigning load addresses to the different program components and adjusting the code in the loaded program, correspondingly. The relocations are the pointers in the code that need to be adjusted depending on where the executable is loaded into memory. An executable file contains a relocation table, which lists the locations of the pointers in the code as offsets from the beginning of the file. The table itself is only needed at load time, and is subsequently discarded.

As it turns out, the combination of the location and the value of pointers naturally provide unique signatures for the pages of an executable file that we can use as a basis for a fingerprint. For that purpose, `codeid` 1) extracts the relocation table from the file; 2) divides it into page-sized blocks; and 3) uses the relocations to create separate signatures for each page. The signature itself consists of a list of offsets to pointers from the beginning of a page, and the corresponding pointer values. Figure 1 illustrates the process. Given a set of `codeid` signatures, the memory capture is split into pages and each one is compared (as explained below) with every page signature in the set. Matches are tallied per known executable to determine the best match.

Table 1: Example: Pointer difference.

Pointer location	In-memory pointer (α)	In-file pointer (β)	Difference
`0x16`	`0xF8CC24E0`	`0x000104E0`	`0xF8CB2000`
`0x3B`	`0xF8CC2490`	`0x00010490`	`0xF8CB2000`
`0x40`	`0xF8CC2500`	`0x00010500`	`0xF8CB2000`
`0x5A`	`0xF8CC2584`	`0x00010584`	`0xF8CB2000`

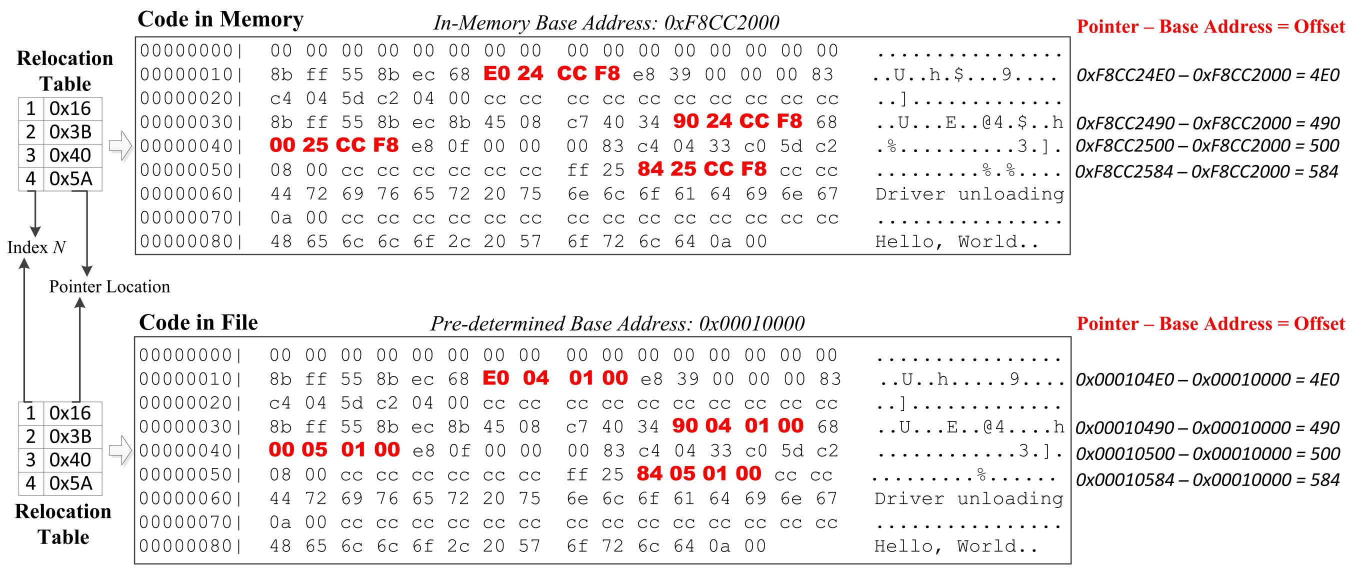
3.3 Design Rationale

As outlined above, signatures are created from the executable file but are compared to live memory pages, and creates several challenges we need to overcome. The first one is ASLR.

When an executable file is created, it is assigned a notional base address and all absolute addresses of pointers in the code are given relative to the base address. If the file is loaded into the memory at the notional base address, then all pointers could be used as is, and there would be a one-to-one correspondance between on-disk and in-memory representations of the code page; the comparison would be trivial. However, due to ASLR, the file is loaded with a randomized base address, rendering a direct comparison meaningless.

Figure 2 illustrates the difference in pointer values when the file is not loaded at its pre-determined base address. It shows two code snapshots of a *hello world* kernel driver for *MS Windows*. The first snapshot is taken from the memory when the driver is loaded at the memory location `0xF8CC2000`. The second snapshot is obtained from the driver file, whose pre-determined base address is `0x00010000`. Since the base addresses are different, the pointer values are also different.

However, as shown on the diagram, if we subtract the pointer values from the base address, the resulting *offset* values are the same in both cases. This means that, instead of directly using pointer values in signatures, we can compute and use offset values in the signatures. To compute


Code in Memory
In-Memory Base Address: 0xF8CC2000
Pointer – Base Address = Offset
Relocation Table
1 0x16
2 0x3B
3 0x40
4 0x5A
Index N
Pointer Location
Code in File
Pre-determined Base Address: 0x00010000
Pointer – Base Address = Offset
1 0x16
2 0x3B
3 0x40
4 0x5A
Relocation Table

```
00000000|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010|  8b ff 55 8b ec 68 E0 24  CC F8 e8 39 00 00 00 83   ..U..h.$...9....   0xF8CC24E0 – 0xF8CC2000 = 4E0
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc   ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 24 CC F8 68   ..U...E..@4.$..h   0xF8CC2490 – 0xF8CC2000 = 490
00000040|  00 25 CC F8 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   .%..........3.].   0xF8CC2500 – 0xF8CC2000 = 500
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 25 CC F8 cc cc   .........%.%....   0xF8CC2584 – 0xF8CC2000 = 584
00000060|  44 72 69 76 65 72 20 75  6e 6c 6f 61 64 69 6e 67   Driver unloading
00000070|  0a 00 cc cc cc cc cc cc  cc cc cc cc cc cc cc cc   ................
00000080|  48 65 6c 6c 6f 2c 20 57  6f 72 6c 64 0a 00         Hello, World..
```

```
00000000|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010|  8b ff 55 8b ec 68 E0 04  01 00 e8 39 00 00 00 83   ..U..h.....9....   0x000104E0 – 0x00010000 = 4E0
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc   ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 04 01 00 68   ..U...E..@4....h   0x00010490 – 0x00010000 = 490
00000040|  00 05 01 00 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   ............3.].   0x00010500 – 0x00010000 = 500
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 05 01 00 cc cc   .........%......   0x00010584 – 0x00010000 = 584
00000060|  44 72 69 76 65 72 20 75  6e 6c 6f 61 64 69 6e 67   Driver unloading
00000070|  0a 00 cc cc cc cc cc cc  cc cc cc cc cc cc cc cc   ................
00000080|  48 65 6c 6c 6f 2c 20 57  6f 72 6c 64 0a 00         Hello, World..
```

Figure 2: Example: ASLR impact on pointer values, and consistency in offsets

the base address, we use a *cross-pointer differential technique*. Since all pointers are derived from the base address, the difference between any pointer's value in its on-disk and in-memory representations is equal to the difference of base addresses of code in file and memory.

The difference of base addresses of kernel module in memory and file (from the last example) is `0xF8CC2000 - 0x00010000` = `0xF8CB2000`. The relocation table has four elements and, as shown in Table 1, the pointer difference is always equal to the difference in base addresses.

More formally, given n relocation in a code page, we denote each relocation in memory by $\alpha(i)$ and in file by $\beta(i)$, respectively ($0 \le i < n$). Then a successful match is equivalent to the following predicate:

$$\forall 0 \le i < n, \alpha(i) - \beta(i) = \alpha(i+1) - \beta(i+1) \qquad (1)$$

Computing base address of in-memory code. By using the relative pointer differences, we avoid the need to know the base in-memory address B_m. However, knowing the base address is useful for other purposes, such as the creation of more robust signatures for data structure (e.g. *EPROCESS*). Therefore, we extend the technique to derive it.

Let B_f be the base address of the in-file code. Then,

$$B_m = \alpha(i) - \beta(i) + B_f, \text{for any } i : 0 \le i < n. \qquad (2)$$

(It should be clear that this technique is *not* an attack on ASLR–an attacker would need access to RAM *before* running the calculation.)

Paging Considerations. The effectiveness of `codeid` is clearly dependent on the number of pages actually present in main memory. For the *MS Windows* kernels we found that about 25% are marked *nonpaged*, which makes them a reliable target. Kernel modules (drivers) are also nonpaged by default. For user applications our observations show that, under normal conditions, the effects are negligible for the applications we observed. (Further work would be needed to study the behavior under memory shortage conditions.)

Algorithm 1 Single page signature match

```
page_match($P, S$) : $boolean$
$diff = uint32(P[S[1].offset]) - S[1].ptr$
for $j = 2$ to $|S|$ do
    if $(uint32(P[S[j].offset]) - S[j].ptr) \neq diff$ then
        return $false$
return $true$
```

We use a *majority-wins* approach, which in practice eliminates (Section 4.4) much of the paging-related noise. One further refinement is to ignore pages marked as discardable (about 13% of the kernel) as they have almost no chance of being in memory.

Correct *page alignment* between in-file and in-memory pages for the executable is another potential problem. Fortunately, our study found that these are always aligned so no additional effort is necessary.

3.4 Signature Comparison

Recall that a (page) signature consists of a list of offset-pointer (O-P) pairs (<*offset*, *ptr*>), and that a file fingerprint consist of a list of such signatures. More formally, let F be a file fingerprint, $|F|$ be the number of page signatures it contains, and $S_k, k = 1..|F|$ be the individual signatures.

Let $|S_k|, k = 1..|F|$ be the length (number of O-P pairs) of each constituent signature, and $S_i[j], i = 1..|F|, j = 1..|S_i|$ be the individual $<offset, ptr>$ O-P pairs. Also, denote by $|M|$ the size of the memory image in pages, and by $P_k, k = 1..|M|$ the content of individual pages (byte arrays), each of which is of size $|P|$. We use the function $uint32(P_k[j]), j = 0..|P| - 3$ to denote the extraction of a 32-bit value from page P_k, starting at location j.

We express the *page_match* predicate from equation (1) between a raw memory *page* P and a *signature* S in the algorithm 1. Thus, a match between fingerprint F consisting of signatures $S_1, ..., S_{|F|}$ and memory image M consisting of pages $P_1, ..., P_{|M|}$ returns the number of signatures matched

Algorithm 2 Count page signature matches

```
match_count($F, M$) : $int$
$count = 0$
for $i = 1$ to $|M|$ do
  for $j = 1$ to $|F|$ do
    if $signature\_match(P_i, S_j)$ then
      $count = count + 1$
return $count$
```

Algorithm 3 Best fingerprint match(es)

```
best_match($RS, M$) : $int$
$m = 0, result = \varnothing$,
$match[1..|RS|] = \{0, ..., 0\}$
for $i = 1$ to $|RS|$ do
  $match[i] = fingeprint\_match(RS_i, M)$
  $m = max(m, match[i])$
for $i = 1$ to $|RS|$ do
  if $match[i] == m$ then
    $result = result \cup RS_i$
return $result$
```

Algorithm 4 Creating a filter table.

```
$filter = new$ hashtable()
for $i = 1$ to $|RS|$ do
  for $j = 1$ to $|S_i|$ do
    $key = S_i[j].offset \mid (uint32(S_i[j].ptr)$ & 0x0FFF)
    $value = filter.lookup(key)$
    if $value ==$ nil then
      $value = new$ set()
    $value = value \cup S_i$
    $filter.put(key, value)$
```

Algorithm 5 Signature matching with O-P filtering.

```
for $i = 1$ to $|M|$ do
  for $j = 1$ to $|P| - 4$ do
    $key = j \mid (uint32(P_i)$ & 0x0FFF)
    $candidates = filter.lookup(key)$
    for $S$ in $candidates$ do
      if $signature\_match(P_i, S)$ then
        $result = result \cup S$
return $result$
```

(Algorithm 2). The result of matching *all* the signatures in a reference set RS would simply return the identities of those signatures that are equal to the highest fingerprint match (Algorithm 3). There are some optimizations that can be done in practice to improve the computation but, fundamentally, the complexity is proportional to $|M| \times |RS|$. Scanning through all of memory is unavoidable (in the general case) but the reference set could potentially be preprocessed to speed up the computation.

The key insight in this respect is the recognition that the least significant 12 bits in a pointer value (for a 4KiB page) remain constant after load-time adjustments (Table 1, Figure 2). One way to take advantage of this property is to create a hashtable that uses the concatination of 12-bit page offset and 12-bit pointer value as a key. The corresponding value includes the set of all signatures that contain that particular O-P combination. During the match process, the table serves as a filter that limits the matching process to signatures that contain the given O-P pair.

Table 2: Test data used in evaluations

Dataset	#1	#2	#3	#4	Total
Files	20	17,010	26	34,014	51,070

The process of creating the filter is illustrated by Algorithm 4, where the "|" sign stands for concatination and the "&" sign for the *bitwise AND* operation. Algorithm 5 shows a modified version of the baseline approach that incorporates filtering. Once the results are obtained, it is trivial to pick out the best match(es).

Conceptually, the filtering does not alter the asymtpotic complexity–in the worst case, all signatures could end up in the same set. However, under the assumption of relatively uniform distribution of the O-P keys (which we have observed empirically), it does reduce the number of signature comparisons by a substantial constant factor. As our experimental results in the following section show, this makes a big difference for larger sets, which is the most important case.

4. EVALUATION

We have developed a proof-of-concept implementation in *C* and used to evaluate our technique. The current version of `codeid` is approximately 1,000 lines of code and works with 32-bit *MS Windows* executables.

4.1 Test Data

We used four datasets (Table 2) with a total of 51,070 executable files: Dataset #1 (*OS Kernels*) contains the kernels for *MS Windows* 2000 Server, XP SP1 (service pack 1), SP2 and SP 3, Vista with SP0 (initial release), SP1, and SP2, Windows 7 SP0 and SP1, and Windows 8 and 8.1. Since we are only evaluating 32-bit Windows executables, we also consider the kernel version (`ntkrnlpa.exe`) that has physical address extension (PAE) support, in addition to `ntoskrnl.exe` that has no PAE support.

Dataset #2 (*MS Windows system libraries and executables*) consists of the application (`.exe`) and library (`.dll`) files of `system32` folder from 11 different versions of *MS Windows* (2000 Server, XP, Vista, 7, and 8).

Dataset #3 (*popular applications*) contains recent versions of eight popular *MS Windows* applications:

Adobe Reader (`AcroRd32.exe`), *AVG antivirus* (`avgui.exe`), *Google Chrome* (`chrome.exe`), *Command Prompt* (`cmd.exe`), *Firefox* (`firefox.exe`), *Internet Explorer* (`iexplore.exe`), *Windows Media Player* (`wmplayer.exe`), *WinRAR Archiver* (`WinRAR.exe`). We also obtained the old versions of the applications[1], installed them, and extracted the executable files that get loaded into memory.

Dataset #4 (*malware*) consists of a sampling of malware executables obtained from the *VX Heaven* public repository [19]. The samples are already split into several different categories–backdoor, constructor, exploit, flooder, packed, rootkit, trojan, virus, and worm.

4.2 Relocation Prevalence & Code Coverage

The first obvious questions to study are the *prevalance* of relocations in executable files–how many relocations *per page*

[1]From `http://www.oldapps.com/`

Table 3: Set #1 (kernels) relocation prevalence and coverage.

Version	Prevalance (P_1)	Coverage (C_1) (%)
2000 Server	66.26	85.90
XP SP1	59.97	88.02
XP SP2	60.12	88.50
XP SP3	56.26	88.08
Vista SP0	56.25	84.61
Vista SP1	55.25	85.99
Vista SP2	55.31	85.79
Win 7 SP0	54.28	86.04
Win 7 SP1	54.05	86.14
Win 8	51.64	91.89
Win 8.1	51.64	91.89

Table 4: Set #2 (system) relocation prevalence and coverage.

Version	Files	P_2	C_2
2000 Server	811	99.77	73.51
XP SP1	923	98.91	72.90
XP SP2	928	94.98	74.39
XP SP3	1,023	95.01	74.25
Vista SP0	1,623	99.66	70.96
Vista SP1	1,641	99.58	70.99
Vista SP2	1,657	99.98	70.60
Win 7 SP0	1,787	101.05	70.63
Win 7 SP1	1,987	101.95	70.21
Win 8	2,259	116.73	71.68
Win 8.1	2,371	116.36	72.43
Weighted Avg		104.12	71.72

Table 5: Set #3 (applications) relocation prevalence and coverage.

Application	P_3	C_3
Adobe Reader 9.4	75.67	7.41
10.1.4	75.66	72.05
11.0.03	79.92	90.60
AVG 2012	119.48	93.37
2013	106.87	94.18
2014	107.04	94.08
Chrome 33.0.1750.146	58.96	66.16
33.0.1750.154	58.98	65.83
34.0.1857.116	57.91	68.56
cmd 6	96.12	70.21
6.1	93.88	90.95
6.2	107.91	83.33
Firefox 23	127.33	4.69
24, 25	78.00	3.17
27, 28	79.00	3.17
IExplorer 7	109.57	9.27
8	142.60	6.21
9	98.33	3.35
10	89.50	2.20
Media Player 11	88.00	5.13
12	108.00	5.13
WinRAR 3.91	108.60	77.22
4.2	105.48	77.65
5.01	99.10	78.45

Table 6: Set #4 (malware) relocation prevalence and coverage.

Category	Files	P_4	C_4
Backdoor	8,654	372.00	71.57
Constructor	106	316.08	66.95
Exploit	140	258.00	66.51
Flooder	136	220.59	72.84
Packed	43	343.00	60.65
Rootkit	562	258.00	60.62
Trojan	22,744	464.60	71.78
Virus	398	310.67	71.88
Worm	1,231	340.00	74.29
Weighted Avg		428.87	71.59

Table 7: Set #1 signature overlap distribution.

Version	0%	20%	40%	60%	80%	99%	**100%**
Win2000	100						
WinXP	100						
WinVista	25.30	72.18	1.89	0.16	0.21	0.26	
Win7	65.91	33.88	0.21				
Win8	100						

can we expect to find–and (code) *coverage*–what fraction of the pages in the executable contain relocations.

Specifically, we define $P_i, i = 1..4$ to be the mean number of relocations per page (for each version/category) of dataset i. Similarly, $C_i, i = 1..4$ is the percentage of pages containing relocations (per version/category).

Tables 3 through 6 present a summary of our findings. In all cases we found the minimum number of relocations, if present, to be at least *four*, which is above the minimum of *two* necessary to build a page signature.

4.3 Collisions & Signature Overlap

Clearly, the accuracy of the `codeid` matching process depends critically on the uniqueness of the page and file signatures. In this section, we study the collision rates of signatures across different executables in our dataset.

Recall that each file signature is a set of page signatures with the latter consisting of a sequence of *offset-relocation* (O-R) pair elements. For the rest of this section, we quantify the level of O-R collisions among page signatures.

Let the total number of O-R pairs in a page signature be r_j and let c_j be the number of collisions (matching pairs in another signature) for page j. We define the *overlap rate* O_j for page j as $O_j = c_j/r_j, j = 1..N$ (N is the total number of pages).

Tables 7 through 10 present the distribution of O-R collisions per page by quantile, as well as the special cases of 0% and 100% collisions (a blank cell indicates zero, while 0.00 means a rate of less than 0.001%). For example, in Table 7 the row for *Vista* shows that 25.3% of the pages had no overlap, 72.2% had between 1 and 20 percent overlap, 1.89% had between 21 and 40 percent overlap, and so on. (The results are aggregated by major version, but *all* pairs of page signatures were compared across the set.)

Recall that, in practical terms, the only collisions that truly matter are the 100% ones–the rest of the columns are presented for completenes. As long as there is at least one

Table 8: Set #2 signature overlap distribution.

Version	0%	20%	40%	60%	80%	99%	**100%**
Win2000	76.66	22.80	0.47	0.04	0.01	0.02	
WinXP1	78.40	21.20	0.36	0.02		0.01	**0.00**
WinXP2	86.27	13.59	0.13		0.00	0.01	**0.00**
WinXP3	94.50	4.67	0.31	0.19	0.14	0.16	**0.03**
WinVista	99.93	0.06	0.01	0.00			
Win7	99.81	0.17	0.01				**0.01**
Win8	99.83	0.17	0.00				**0.00**

Table 9: Set #3 signature overlap distribution.

Application	0%	20%	40%	60%	80%	99%	**100%**
AVG 2012	12.68	61.97	21.75	3.50	0.11		
2013	13.77	69.96	12.15	4.01		0.11	
2014	100						
Chrome							
33.0.1750.146	38.93	61.07					
34.0.1857.116	36.84	63.16					
IExplorer 7	42.86	50.00	7.014				
8	100						
9	100						
10	50.00	50.00					

Table 10: Set #4 signature overlap distribution.

Category	0%	20%	40%	60%	80%	99%	**100%**
Backdoor	61.02	10.47	00.48	00.17	00.16	00.54	**27.16**
Constructor	98.75	01.20	00.01	00.03		00.01	
Exploit	88.42	11.29	00.24		00.02	00.02	
Flooder	96.29	02.77	00.10	00.12	00.20	00.39	**00.14**
Packed	92.11	07.89					
Rootkit	95.14	03.12	00.07		00.09	00.56	**01.01**
Trojan	79.80	10.48	00.72	00.34	00.32	00.63	**07.70**
Virus	95.17	04.64	00.02		00.01	00.02	**00.13**
Worm	99.45	00.24	00.04	00.07	00.10	00.07	**00.03**

unique O-R pair in the page signature, the *page_match* predicate (Algorithm 1) would return *false* so no confusion will take place.

For the set #1 experiments (Table 7), we generated signatures for a total of 20 kernel files (both with and without PAE support), resulting in 11,472 page signatures and 641,636 O-R pairs. Clearly, the main result is that no two pages completely overlap; that is, *all page signatures are unique*. Further, on most versions, there is no O-R overlap at all.

Set #2–17,000 *system* files–present a very similar picture (Table 8), with the additional trend of decreasing overlap from older to newer versions. We are not in a position to definitively explain the trend; one possibility is that newer compiler optimizations lead to less stable O-R configurations in response to minor code changes.

Popular applications (set #3) further confirm the uniqueness of relocations. Five out of the eight applications–*Adobe Reader*, *cmd*, *Firefox*, *Media Player*, and *WinRAR* contain *only non-overlapping* page signatures. This is a particularly important property for applications like *Firefox* and *Media Player* that have only 2-5 page signatures per file. It is notable also that the five *Firefox* versions cover a release period of only eight months (Aug 2013–Apr 2014). The overlap for the remaining three applications (Table 9) stays almost completely in the lowest quantile.

Although malware detection/classification is *not* the main target application of this work `codeid` can do a thorough job of finding *known* malware executables. Table 10 shows a summary of our study of set #4 (976,754 page signatures). Almost all the signatures are quite distinct; the only major exceptions are the *backdoor* and (to a lesser degree) *trojan* categories, where full signature overlap is observed in 27.12% and 7.67% of the cases, respectively.

Using similarity hashing [14] and the `sdhash` tool [16] on the files reveals that the *backdoor* set contains 964 closely related versions of `Backdoor.Win32.Hupigon` (similarity score 95 out of 100), and it is the main reason for signature collisions. Using the same approach we find that `Trojan-Downloader.Win32.Banload` contributes 718 near-identical versions accounting for 4.4% of the samples (and collisions) in the category.

In summary, our collision study finds ample evidence for the assertion that page relocation tables provide a set of characteristic attributes for almost any *MS Windows* executable. The only exceptions we found are binaries that are nearly identical, as determined by static analysis.

4.4 Accuracy Analysis

In this section, we measure the effectiveness of `codeid` on page and file level in both kernel and user space against actual memory captures. We use VMware's VM snapshot mechanism to obtain the RAM targets, which are 2GB in size.

4.4.1 Page-level accuracy

Our first study quantifies the false positive (FP) and false negative (FN) rates for page signatures. For that purpose, we need a method for establishing the ground truth, and due to the uncertainty of paging, simply loading/running the binary is not enough to assume that all pages would be in memory. We need a method to *ascertain* which *individual* pages are *physically* present in memory.

Establishing ground truth. To establish a baseline, we analyze the memory image using *libvmi* [7, 11] library (which is an outgrowth of the earlier work on XenAccess [1]). Unlike our approach, *libvmi* finds and interpretes the relevant kernel data structures, such as `LDR_DATA_TABLE_ ENTRY`, `EPROCESS`, `PEB`, and `PEB_LDR_DATA`. Using this information, it can identify the virtual base address and size of kernel, as well as other executables, including dynamic link libraries (`.DLL`), applications (`.EXE`), and kernel modules (`.SYS`).

Specifically, we employ *libvmi* to map the running processes to a set of physical pages. Using the map, we define specific page targets for `codeid`; that is, we expect the tool not only to show a match for the correct executable, but also to point to a page owned by the respective process.

Under this definition of the ground truth, we define the four possible experimental outcomes as follows:

- *True Positive* (TP): `codeid` has identified the correct (memory page) signature *and* the page belongs to the correct process.
- *False Positive* (FP): `codeid` has identified the wrong signature *or* the page does *not* belong to the correct process.
- *True Negative* (TN): `codeid` yields *no* match *and* the memory image does *not* contain the target binary.
- *False Negative* (FN): `codeid` yields *no* match *and* the memory image *does* contain the target binary.

Table 11: Page-level accuracy: kernel modules

Version	Pages	FPR	FNR	Accuracy
XP1	1,517	0.0000	0.0000	1.0000
XP2	2,001	0.0020	0.0000	0.9980
XP3	1,922	0.0000	0.0000	1.0000
Vista0	3,267	0.0083	0.0000	0.9917
Vista1	3,342	0.0009	0.0000	0.9991
Vista2	3,454	0.0026	0.0000	0.9974
Win7	3,790	0.0000	0.0000	1.0000
Win7.1	3,717	0.0016	0.0000	0.9984
Win8	5,193	0.0029	0.0000	0.9871
Win8.1	5,336	0.0002	0.0000	0.9998
Overall	33,539	0.0019	0.0000	0.9981

Table 12: Page-level accuracy:non-kernel processes

Version	Pages	FPR	FNR	Accuracy
XP1	4,992	0.0048	0.0000	0.9952
XP2	5,962	0.0002	0.0000	0.9998
XP3	5,359	0.0011	0.0000	0.9989
Vista0	14,298	0.0034	0.0000	0.9966
Vista1	15,117	0.0064	0.0000	0.9936
Vista2	15,382	0.0000	0.0000	1.0000
Win7	16,232	0.0025	0.0000	0.9975
Win7.1	17,426	0.0007	0.0000	0.9993
Win8	26,128	0.0010	0.0000	0.9990
Win8.1	28,452	0.0039	0.0000	0.9961
Overall	149,348	0.0024	0.0000	0.9976

As it turns out, the criteria are in need of some refinement. The first adjustment is for FP results–in addition to the page mapped to the address space of the running process, `codeid` would also find the original page in the file cache (if present) regardless of whether the process is still running. (From the point of view of `codeid`, it looks as if the binary was loaded exactly at the nominal base address.) This special case can be detected by simply keeping a crypto hash of the page, and–depending on use case–turn hash-based page matches on/off. Therefore, we exclude identical page matches from the FP results.

The second adjustment is necessary for classifying FN outcomes. Upon inspecting preliminary results, we realized that in some cases, although the page is mapped, it contains only `0x00` or `0xFF` bytes. Clearly, such an outcome is a *true* negative–although it trivially satifies equation (1)–so we filter out from the FN set all pages with precisely zero (Shannon) entropy.

For these experiments, we split all the processes found in the memory capture into kernel and non-kernel, and Tables 11 and 12 summarize our respective observations. The first column shows the *MS Windows* version, followed by the total number of pages extracted, the FP rate, the FN rate, and the standard measure of accuracy:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

The first important observation is that the there are *zero false negatives.* This is to be unexpected–if the page is indeed in memory, it must satisfy the *page_match* predicate.

The presented false positive rate is a conservative estimate and can be further improved–manual examination of a sampling of the false positive results indicate that they are triggered primarily by low (but not zero) entropy pages that could further be filtered out.

Overall, the accuracy–which in the absense is false positives is equal to the *TPR*–is consistantly high across both sets. By aggregating the results, we conclude that the overall *page-level* accuracy is 0.9977.

4.4.2 File-level accuracy

For evaluating the file-level accuracy, which would be the user-experienced performance, we considered two case studies–kernel modules and applications.

Kernel accuracy. First, we focus on the classification error across 10 different versions of the kernel modules. The main rationale here is that, unlike other executables, we can clearly attribute the modules to different versions.

There are a total of 182 different (by name) modules across the ten *MS Windows* distributions. Of these, only 26 are ideal targets as they are present in *all* versions. Since that is too small a set, we expanded it to include all modules that occur in at least three versions, which brings our evaluation set to 157.

Table 13 shows the results of matching the memory images (columns) vs. the corresponding file-derived `codeid` signatures (rows); blank cells indicate zeroes. The *GT* column represents the ground truth–the total number of binaries present in the image, as per kernel data structures. The *TPR* column is the TP rate–the ratio of the diagonal element to GT. Thus, the ideal confusion matrix would consist of the diagonal elements matching the corresponging *GT* value, and all other elements set to zero.

The evaluation procedure emulates what our tool does–for each file fingerprint, we match all the page signatures against the memory capture and count the total number of matches. All fingerprints that have signature matches equal to the maximum number of matches are declared a file match. Since more than one version may match the file, row numbers may add up to more than the *GT*.

Based on page-level results, we would expect near-perfect true positive rates across the sets; indeed, with one exception, the rates range from 0.95 to 1.0. The difference in performance is accounted for by differences in the fraction of pages *actually loaded* by the page system. The overall fraction of pages not loaded for all 10 images was only 2.2%–773 out of 33,539; however, the *VistaSP1* image alone had 434 pages (out of 3,342) missing, or 13%. (The latter is probably the result of a snapshot taken too soon after boot.)

Further examination showed that missing pages tend to be clustered, typically in the form of entire modules having no valid pages in memory. Thus, their effects are amplified in our statistics (e.g., a missing module with a 3-page signature represents only 0.1% of the page signatures for *VistaSP1* but a full 1% of the number of kernel modules).

Conceptually, such lapses are not the fault of `codeid` (or any other comparable method) simply because the data to be detected is not present. This underscores the importance of page-level detection, and suggests that page-level performance is a cleaner measure of algorithmic potential, as it excludes the (unpredictable) effects of paging.

The use of *VMware* memory snapshots also presented us with an additional (unplanned) test case–the detection of

Table 13: Kernel modules confusion matrix

	WinXP			Vista			Win7		Win8		GT	TPR
WinXP	**60**	10	5								61	.9836
	9	**79**	19								82	.9634
	5	20	**74**								74	.9600
Vista				**95**	7						95	1.000
				21	**75**	45					97	.7732
				15	19	**95**					95	1.000
Win7							**102**	57			102	1.000
							58	**103**			103	1.000
Win8									**110**		110	1.000
										109	109	1.000

Table 14: *Firefox* 23–29 confusion matrix

Version	23	24	25	26	27	28	29	GT
Firefox 23	3							3
24		2						2
25			2	2				2
26			2	2				2
27					2	1		2
28					1	2	1	2
29						1	2	2

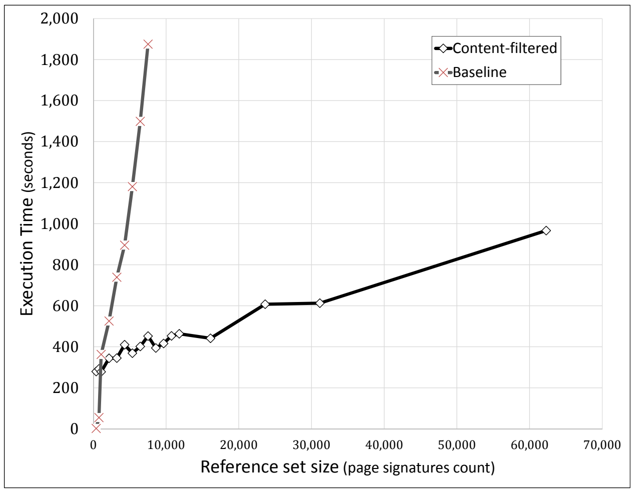


Figure 3: Algorithm speed performance: baseline vs. content-filtered

VMware's own drivers in the image. Since they are identical across all cases, the matches were excluded from Table 13. However, in the initial results, they appeared as background noise in the form of 2-5 additional file matches across *all* cases. The result were illustrative in that page detection per module was either 100%, or 0%, which corresponds to the driver being either loaded, or not.

Application accuracy. In the previous section, we saw that applications tend to have no page signature collisions, which makes the testing of file-level matches a rather trivial pursuit. Therefore, we focused on testing across the different versions of a single application. In an attempt to construct a difficult target, we picked seven consecutive version of *Firefox*–23 through 29–and performed the same analysis as with the kernel case. Recall that *Firefox* has no more than three pages with relocations, and individual releases come every six weeks. Taken together, these present among the most difficult challenges, as we would expect the executable to change only incrementally and a large amount of commonality to be present across the binaries.

Nonetheless, the results in Table 14 show that `codeid` is successful in identifying *all* the pages of the correct executable, and only two neighboring versions (25 and 26) are tied.

4.5 Throughput

Recall that in Section 3.4 we proposed two versions of the fingerprint matching algorithm. Both scan the given memory image page-by-page and match them against the reference set of signatures obtained from the files. The difference between the two is that the first one compares all signatures with each page of memory, whereas the second one scans memory byte-by-byte, reading 32-bits at a time, and identifies more relevant signatures that can be matched with the page. Thus, we would expect the second approach to be more efficient for working with a large number as it shifts the *primary* dependence of processing time from the number of signatures to the size of pages.

We ran our PoC implementations on a 2.6GHz Intel Core i7 CPU using a 2GB target (33,554,432 pages), and normalized results to represent time per GB of RAM processed. Initially we use Windows XP SP2 memory dump to search the page signatures of *ntoskrnl.exe* (381 in number) and then later start using the 1,074 signatures of Windows 8.1 kernel.

Figure 3 depicts the throughput of the algorithms as a function of the number of signatures. As we would expect, processing time for both algorithms grows linearly. The baseline version starts much lower but rises at a much steeper angle than the second algorithm, which filters the candidate signatures before comparison; the crossover point is around 1,000 signatures.

Another way to look at the relative performance it to consider throughput in terms of of page-signature matches per second that each version can perform. For the baseline, the relevant number is 2.1–2.5 million for sets of 1,000+, whereas the content-filtered version starts at a comparable 2 million for the 1,074-signature set but steadily rises to 33.8 million for the last set with 62,292 signatures. We should emphasize that our reference implementation is not, of yet, optimized for throughput. There are numerous opportunities to speed up the processing, such as filtering out of pages based on location and/or content, sampling of the signatures, further indexing of the reference set, cache optimization, and concurrent processing. These are beyond the scope of this work, which is primarily targeted at evaluating feasibility and suitability of the proposed technique.

4.6 Comparison with prior work

Recall that prior work focuses on the *specific* problem of identifying OS kernels, and not of identifying code in general. Indeed, the approaches taken have focused on kernel specific privileged instructions, the content of kernel data structures, and special CPU registers, and *cannot* be extended to applications. Thus, for an apples-to-apples comparison, we limit ourselves to identifying kernel versions.

Particularly, we used *MS Windows* kernels for evaluation since they have relocatable code. The Linux kernel, on the other hand is PIE and does not lie within the scope of the paper. Furthermore, we have already argued that finding PIE code is not a difficult problem as we can exactly compute

Table 15: Head-to-head comparison of `codeid` and *OS-Sommelier*

Windows Version	*OS-Sommelier*		`codeid`	
	VMware	QEMU	VMware	QEMU
Win Server 2000	✓	✓	✓	✓
Win XP SP1	×	×	✓	✓
Win XP SP2	✓	×	✓	✓
Win XP SP3	✓	✓	✓	✓
Win Vista SP0	✓	✓	✓	✓
Win Vista SP1	✓	✓	✓	✓
Win Vista SP2	×	×	✓	✓
Win 7 SP0	✓	✓	✓	✓
Win 7 SP1	×	×	✓	✓
Win 8	×	×	✓	✓
Win 8.1	×	×	✓	✓

hash of code pages (from the kernel image file) and find them in RAM. Indeed, such an approach would be much simpler and more robust than prior work, and would also cover any other PIE code.

In [3], the authors have performed an extensive evaluation showing that *OS-Sommelier*'s detection capabilities are strictly better than prior approaches based on profiling the kernel implementation using CPU (special) register values, and the contents of the IDT [2].

Therefore, we consider *OS-Sommelier* to be the best representation of the prior state of the art, and the most relevant benchmark for our own work. The authors gracefully provided us with their code so we could perform a head-to-head comparison.

We used both QEMU [12] and VMware to perform the evaluation of `codeid` and *OS-Sommelier* in order to see how sensitive they are. Recall that *OS-Sommelier* needs a memory snapshot of the kernel to generate its signature. We used one set of RAM captures for the signature generation phase, and a different one to perform the actual experiments. Table 15 summarizes the results. As the results show, `codeid` performed perfectly, whereas *OS-Sommelier* encountered a variety of problems, eventually succeeding on only five of the eleven versions tested. Below we briefly summarize our experience with *OS-Sommelier*.

Positive results. The tool was able to successfully generate and recognize both QEMU and VMware images for the following versions: Windows Server 2000, Windows XP SP1 and SP3, Windows Vista SP0 and SP1, Windows 7 SP0.

Inconsistent results. *Windows XP SP2*: The tool was unable to disassemble the pages extracted from QEMU image, which resulted in an "Unknown OS" error both during generation and comparison attempts. However, image from VMware causes no such problems.

Vista SP2 If QEMU image is used to generate signatures, then a VMware-acquired image always gets identified as Vista SP1. Similarly, if VMware image is used as the base, then the corresponding QEMU gets misidentified as Vista SP1.

No results. The tool was unable to generate signatures for Windows 7 SP1, Windows 8, and 8.1. The specific cause of the failure is that the tool cannot find a specific byte pattern–`0f 20 d8 0f 22 d8`–which leads to a crash.

The main takeaway is that the approach taken by *OS-Sommelier* is too complicated and inherently fragile; it needs human input with every new version of the OS. Even more problematic is the high sensitivity to the hypervisor as in an IaaS environment, multiple hypervisor options are the norm.

In constrast, our approach is robust since it only needs the file of the executable (no memory image) to generate the signature and works seemlessly across kernel and application versions. Our technique looks at the code as data and its performance would not be affected by minute details on how the image was acquired. Our method generates perfectly unique signatures for the kernels (Table 7) and detects the exact kernel versions with 100% accuracy.

5. CONCLUSIONS

In this work we considered the problem of identifying known executable code in memory images and proposed a new solution based on using relocation tables as the key identifying characteristic. We showed that relocation tables tend to be quite distinct and, therefore, are an excellent basis for building a unique fingerprint. We demonstrated how the in-file and in-memory version of the pages with relocations can be related as they get transformed by ASLR. In the process, we developed a simple method to calculate the base address of executable.

Unlike prior work, which relies heavily on deep manual analysis and results in fragile methods that are not guaranteed to work on newer versions, *CodeIdentifier* presents a fully automated solution that is fast, accurate, and robust. Our approach is not narrowly focused on kernel version identification but works for any executable, with the kernel being a special use case; the only input required to generate a signature is the file containing the executable. No knowledge of kernel data structures, or any interpretation of the memory capture is necessary.

Our experimental evaluation showed that we can pinpoint individual memory pages as belonging to a known executable with *zero false negatives* and with 99.77% accuracy. We can find trace evidence of prior executions in the file cache and can distinguish them from pages belonging to active processes. *CodeIdentifier* identified perfectly all 11 kernels tested and, in addition, can correctly map kernel modules to their respective *MS Windows* version with a TP rate between 0.96 and 1.00. We showed that our method performs well in distinguishing application with small signatures and close versions. Specifically we demonstrated successful detection and identification of seven consecutive versions of *Firefox*–the most difficult application in our test set.

We have developed a scalable page-signature matching algorithm, which can perform 33.8 million page-signature matches per second (on a sigle core) with a reference set containing over 62,000 page signatures. We expect future work to speed this by at least a factor of 10 by preprocessing the reference set and the memory image.

The main limiting factor to the presented method is the unpredictability of the paging system; our experience shows that, under normal workloads, this is not a notable impediment. Most importantly, the zero false negative rate of our method ensures that, if the target code is in memory, it will be found.

Finally, although not specifically targeted at malware detection, `codeid` can be reliably used for ad-hoc malware signature generation and in-memory scans. This is useful for newly discovered samples (during incident response) before a more succinct signature is derived and incorporated into the security monitoring infrastructure.

6. REFERENCES

[1] W. L. Bryan D. Payne Martim D. P. de A. Carbone. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference*, 2007.

[2] 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, New York, NY, USA, 2009. ACM.

[3] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. OS-Sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 5:1–5:13, New York, NY, USA, 2012. ACM.

[4] *imageinfo*. `https://code.google.com/p/volatility/wiki/CommandReference#imageinfo`.

[5] N. L. P. Jr., A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006.

[6] *libguestfs*. `http://libguestfs.org/`.

[7] *libvmi*. `http://code.google.com/p/vmitools/`.

[8] 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 *NDSS*, 2011.

[9] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, third edition, 2010.

[10] *nmap*. `http://nmap.org/`.

[11] B. D. Payne. Simplifying virtual machine introspection using libvmi, 2012. Sandia Report SAND2012-7818, `http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf`.

[12] *qemu*. `http://qemu.org`.

[13] N. A. Quynh. Operating system fingerprinting for virtual machines. In *DEFCON 18*, 2010. `http://www.defcon.org/images/defcon-18/dc-18-presentations/Quynh/DEFCON-18-Quynh-OS-Fingerprinting-VM.pdf`.

[14] V. Roussev. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*, pages 207–226. Springer, 2010.

[15] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals: Including Windows Server 2008 and Windows Vista*. Microsoft Press, fifth edition, 2009.

[16] *sdhash*. `http://sdhash.org`.

[17] *virt-inspector*. `http://libguestfs.org/virt-inspector.1.html`.

[18] *Volatility*. `https://code.google.com/p/volatility/`.

[19] *VX Heaven*. `http://vxheaven.org`.

[20] *Xprobe2*. `http://sourceforge.net/projects/xprobe/files/xprobe2/`.