Reliable In-Memory Code Identification Using Relocatable Pointers

Irfan Ahmed, Vassil Roussev, Aisha Ali Gombe

Department of Computer Science
University of New Orleans

Contents

- Code fingerprinting
- Problem Statement
- Codeid – proposed approach
  - Challenges – ASLR, paging, and page alignment
- Evaluation
  - Prevalence
  - Coverage
  - Collision
  - Accuracy
- Conclusion

Why Code Fingerprinting?

- Used by several proactive security monitoring, and reactive forensic analysis applications
  - Most incident response and deep forensics techniques requires exact code version being executed on target system
  - Fingerprinting of Malware is at the core of antivirus applications
- In Infrastructure-as-a-service (IaaS), code fingerprinting is a critical tool that enables a large number of automated services
  - Patch management
  - Security services such as code integrity checking

Code Identification

- Network fingerprinting tools
  - such as *nmap* and *xprobe2*
  - Remotely identify kernel versions based on the packets being exchanged
  - inherently unreliable
- Disk filesystem
  - such as *virt-inspector* using *libguestfs*
  - Limited access to non-volatile media such as encrypted disk in the cloud
- Hardware
  - CPU register states containing pointers to low-level data structures such as IDT and GDT to identify kernel versions
  - Does not work on many MS Windows kernels [1]

[1] Y. Gu, et al., Os-sommelier: Memory-only operating system fingerprinting in the cloud, Third ACM Symposium on Cloud Computing, SoCC '12, pages 5:1-5:13, New York, NY, USA, 2012. ACM.

Code Identification

- Physical memory
  - Use cryptographic hash of interrupt handler code as unique a unique feature to identify kernel versions
  - Data structure definitions vary across OS version, which are used for kernel version identification
  - OS-Sommelier
    - Search entire memory dump and identifies the page global table
    - Find kernel page in virtual address space in memory
    - Generate the signature of the kernel, which is cryptographic hash values of kernel pages
    - Same step is performed on target image to identify the kernel version

Problem Statement

- Given a physical memory dump, how we can identify the presence of a known piece of code in the dump, which is running at an arbitrary location
- Goals:
  - **Accurate** – precise results even for closely related code
  - **Robust** – least dependence on in-memory data structures
  - **Performance** – fast enough to be of practical use of scanning live VMs
  - **Fully Automated** – requires no human in the loop

A simple approach

- Divide the executable file into memory size pages
- Compute hash of each page
- Compute the hash of in-memory pages
- and, compare them with the hash values of executable file pages
- Works on Position Independent Code that does not change when loaded into the memory

Does not work on relocatable code

Relocatable code

Code in Memory *In-Memory Base Address: 0xF8CC2000*

```
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.$...9....
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc  ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 24 CC F8 68  ..U...E..@4.$..h
00000040|  00 25 CC F8 e8 0f 00 00  00 83 c4 04 33 c0 5d c2  .%..........3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 25 CC F8 cc cc  .........%.%....
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..
```

Code in File *Pre-determined Base Address: 0x00010000*

```
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....
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
00000040|  00 05 01 00 e8 0f 00 00  00 83 c4 04 33 c0 5d c2  ............3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 05 01 00 cc cc  .........%......
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..
```

Focus of the work

- How to efficiently identify relocatable code
- Consider only 32-bit Windows executables for experiments

Proposed Approach – codeid

Index *N*

Pointer Location

1	0x16
2	0x3B
3	0x40
4	0x5A

Relocation Table

Code in File *Pre-determined Base Address: 0x00010000*

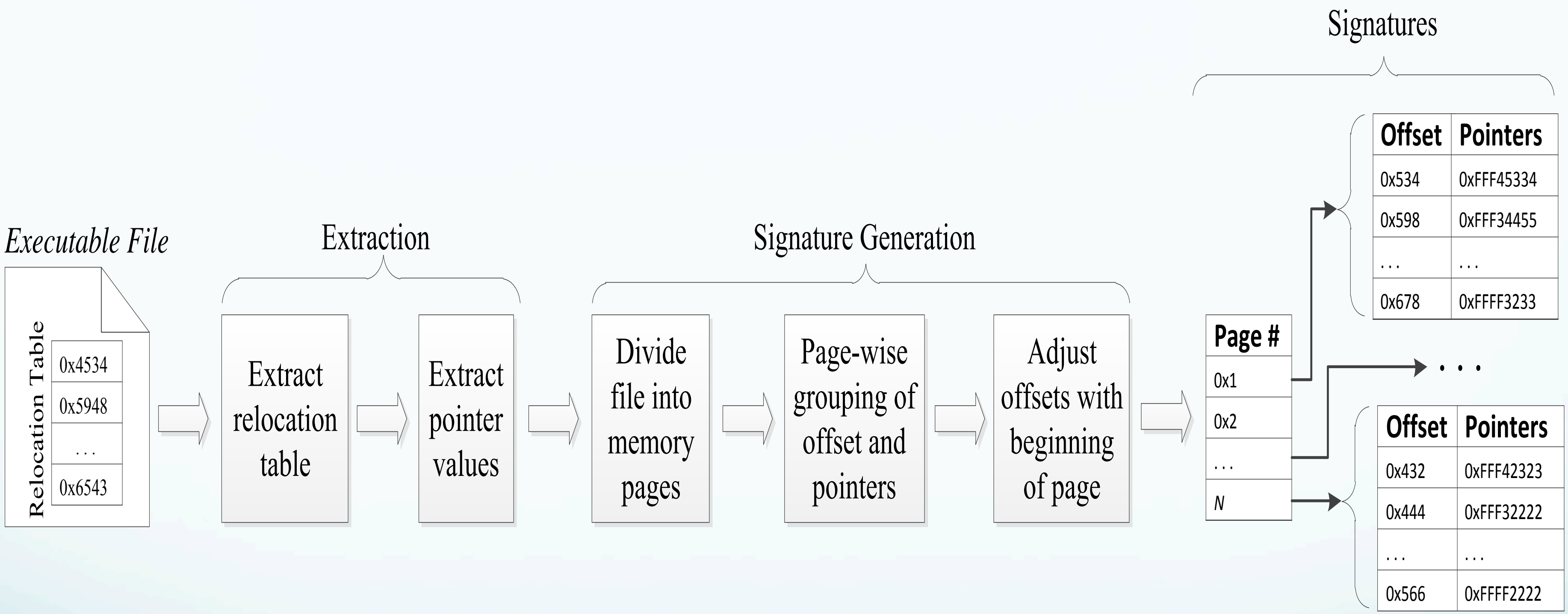
```
00000000|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010|  8b ff 55 8b ec 68 E0 04  01 00 e8 39 00 00 00 83   ..U..h.....9....
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc   ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 04 01 00 68   ..U...E..@4....h
00000040|  00 05 01 00 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   ............3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 05 01 00 cc cc   .........%......
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..
```

page signature ← offsets in relocation table & pointers

Observation (from our study):

Location and pointer values naturally provide unique signatures for the pages of an executable files

Proposed Approach – codeid



ASLR & pointer values

Code in Memory *In-Memory Base Address: 0xF8CC2000*

Relocation Table

1	0x16
2	0x3B
3	0x40
4	0x5A

```
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.$...9....
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc   ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 24 CC F8 68   ..U...E..@4.$..h
00000040|  00 25 CC F8 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   .%..........3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 25 CC F8 cc cc   .........%.%....
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..
```

Pointer – Base Address = Offset

0xF8CC24E0 – 0xF8CC2000 = 4E0

0xF8CC2490 – 0xF8CC2000 = 490

0xF8CC2500 – 0xF8CC2000 = 500

0xF8CC2584 – 0xF8CC2000 = 584

Index *N*

Pointer Location

Code in File *Pre-determined Base Address: 0x00010000*

```
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....
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
00000040|  00 05 01 00 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   ............3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 05 01 00 cc cc   .........%......
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..
```

Pointer – Base Address = Offset

0x000104E0 – 0x00010000 = 4E0

0x00010490 – 0x00010000 = 490

0x00010500 – 0x00010000 = 500

0x00010584 – 0x00010000 = 584

1	0x16
2	0x3B
3	0x40
4	0x5A

Relocation Table

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

on-disk vs in-memory pointer values

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

Base in-memory address (B_m) computation:

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

Paging Considerations

- Not all pages of an executable (kernels/ applications/libraries) are present in main memory
  - About 75% of MS Windows kernels are pageable
  - Application executables are completely pageable
- *majority-wins* approach is used, which in practice eliminates much of the paging-related noise
- Use all pages of executable that are *read-only* and *not discardable*

Correct page alignment

- Alignment between in-file and in-memory pages for the executable
- Enough information is present in the headers of an executable for correct alignment

Test dataset

- Everything hinges on the signatures from relocation tables being *unique*

- Test sets
  1. Kernels
  2. System executables (system32)
  3. Applications
  4. Malware

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

Analysis of the Data

- **Prevalence**: how many relocations per page can we expect to find?
- **Coverage**: what fraction of the pages in the executable contain relocations?
- **Collision**: What are the collision rates of signatures across different executables
- **Accuracy**:
  - *Page Level*: What are the false positive (FP) and false negative (FN) rates for page signatures?
  - *File Level*: Can we just use relocatable code to identify code version?

prevalence & coverage: kernels

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

prevalence & coverage: system

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

prevalence & coverage: applications

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	78.00	3.17
	25	78.00	3.17
	27	79.00	3.17
	28	79.00	3.17
IExplorer	7	109.57	9.27
	8	142.6	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

prevalence & coverage: malware

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

(page) signature overlap: kernels

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						

- 20 kernel files
  - 11,472 signatures and
  - 641,636 Offset-Relocation (O-R) pairs
- All signatures are unique
  - No two pages completely overlap

(page) signature overlap: system

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

- 17,000 system files
  - 667,299 signatures
  - 67,031,628 O-R pairs
  - Similar results from Kernels
- Decreasing overlap from older to newer versions
  - No definite answer
  - One possibility is that newer compiler optimization leads to less stable O-R configurations in response to minor code changes

(page) signature overlap: applications

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					

- 8 popular applications – 26 total versions
  - 5 out of 8 contain only non-overlapping signatures
  - Adobe Reader, cmd, Firefox, Media Player, and WinRAR
  - Five Firefox versions cover a release period of only eight months (Aug 2013 – April 2014)
- The overlap of the remaining three applications stays almost completely in lowest quantile

(page) signature overlap: malware

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

- 9 malware types
  - 34,014 malware samples
  - 976,754 page signatures
  - 124,239,916 pairs
  - Almost all signatures are distinct
- Exceptions: *Backdoor*, and *Trojan*

(page) signature overlap: malware

- Backdoor set contains 964 closely related version of `Backdoor.Win32.Hupigon`
  - sdhash similarity score is 95 out of 100
- Trojan set contains 718 near-identical versions of `Trojan-Downloader.Win32.Banload`
  - Contributing 4.4% of the samples in collision

Page- and File-level accuracy

- Coverage, Provenance, and Overlap are measured on the signatures created from executable files
- Accuracy is measured on memory dumps
- Measuring ground truth
  - LibVMI – an alternate way to find pages in memory dump
  - Unlike codeid, libVMI finds and interprets the data structures such as `LDR_DATA_TABLE_ENTRY`, `PEB_LDR_DATA`, `EPROCESS`, and `PEB`.
  - It identifies the virtual base address and size of kernel and other executables including DLLs, EXEs, and SYS.
  - Pages identified by LibVMI are the target for codeid

Page- and File-level accuracy

- In some cases, page contains only `0x00` or `0xFF`.
  - It satisfies the equation for signature matching
  - Trivially filter out with precisely *zero* entropy
  - Filtering improves false positives
- Pages with very low (but not zero) entropy triggers false positives
- Overall codeid has zero false negative rate, and false positive rate of around 0.0021
  - Page-level accuracy is 99.79%

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

TP: Identify correct page, belong to correct process
FP: Wrong page or page belongs to wrong process
TN: No match and memory image does not contain target binary
FN: No match and memory image contains target binary

page-level accuracy: processes and .dlls

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

TP: Identify correct page, belong to correct process
FP: Wrong page or page belongs to wrong process
TN: No match and memory image does not contain target binary
FN: No match and memory image contains target binary

File-level accuracy: kernel modules

	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

- 182 different modules (by names)
  - across 10 MS Windows versions
- Blank cells indicate *zero* in confusion matrix
- *GT* represents ground-truth
  - Total number of binaries present in memory image

File-level accuracy: kernel modules

- Paging affects the accuracy
  - 2.2% (or 737 out of 33,539) of pages are not loaded in 10 images
  - Vista-SP1 image alone had 434 pages (out of 3,342) missing
    - 13% of pages are missing
    - It is probably the result of the snapshot taken too soon after boot
- VMware drivers are found in all cases

File-level accuracy: Firefox 23 – 29

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

- Applications tend to have *no* page signature collisions
- Challenging case: 7 consecutive versions of Firefox, individual release comes every 6 weeks
  - Coverage of around 3.5%
  - All pages are correctly identified
  - Only two neighboring versions are tied

Signature Comparison—Baseline algorithm

Algorithm 2 Baseline Algorithm

$M \leftarrow$ Number of pages in memory
$N \leftarrow$ Number of executable files
$S_N \leftarrow$ Number of page signatures of an executable
$count = 0$
for $i = 1$ to $|M|$ **do**
  for $j = 1$ to $|F|$ **do**
    for $k = 1$ to $|S_N|$ **do**
      if $signature\_match(P_i, S_{j,k})$ **then**
        $count_j++$
return $count$

Basis of content-filtered algorithm

Code in Memory *In-Memory Base Address: 0xF8CC2000*

Relocation Table

1	0x16
2	0x3B
3	0x40
4	0x5A

```
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.$...9....
00000020|  c4 04 5d c2 04 00 cc cc  cc cc cc cc cc cc cc cc   ..].............
00000030|  8b ff 55 8b ec 8b 45 08  c7 40 34 90 24 CC F8 68   ..U...E..@4.$..h
00000040|  00 25 CC F8 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   .%..........3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 25 CC F8 cc cc   .........%.%....
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..
```

Pointer – Base Address = Offset

0xF8CC24E0 – 0xF8CC2000 = 4E0

0xF8CC2490 – 0xF8CC2000 = 490

0xF8CC2500 – 0xF8CC2000 = 500

0xF8CC2584 – 0xF8CC2000 = 584

Index *N*

Pointer Location

Code in File *Pre-determined Base Address: 0x00010000*

```
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....
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
00000040|  00 05 01 00 e8 0f 00 00  00 83 c4 04 33 c0 5d c2   ............3.].
00000050|  08 00 cc cc cc cc cc cc  ff 25 84 05 01 00 cc cc   .........%......
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..
```

Pointer – Base Address = Offset

0x000104E0 – 0x00010000 = 4E0

0x00010490 – 0x00010000 = 490

0x00010500 – 0x00010000 = 500

0x00010584 – 0x00010000 = 584

1	0x16
2	0x3B
3	0x40
4	0x5A

Relocation Table

Last 12 bits of a pointer are consistent across file and memory

$$key = S_i[j].offset \mid (uint32(S_i[j].ptr)\ \&\ \mathtt{0x0FFF})$$

Signature Comparison—Content-filtered algorithm

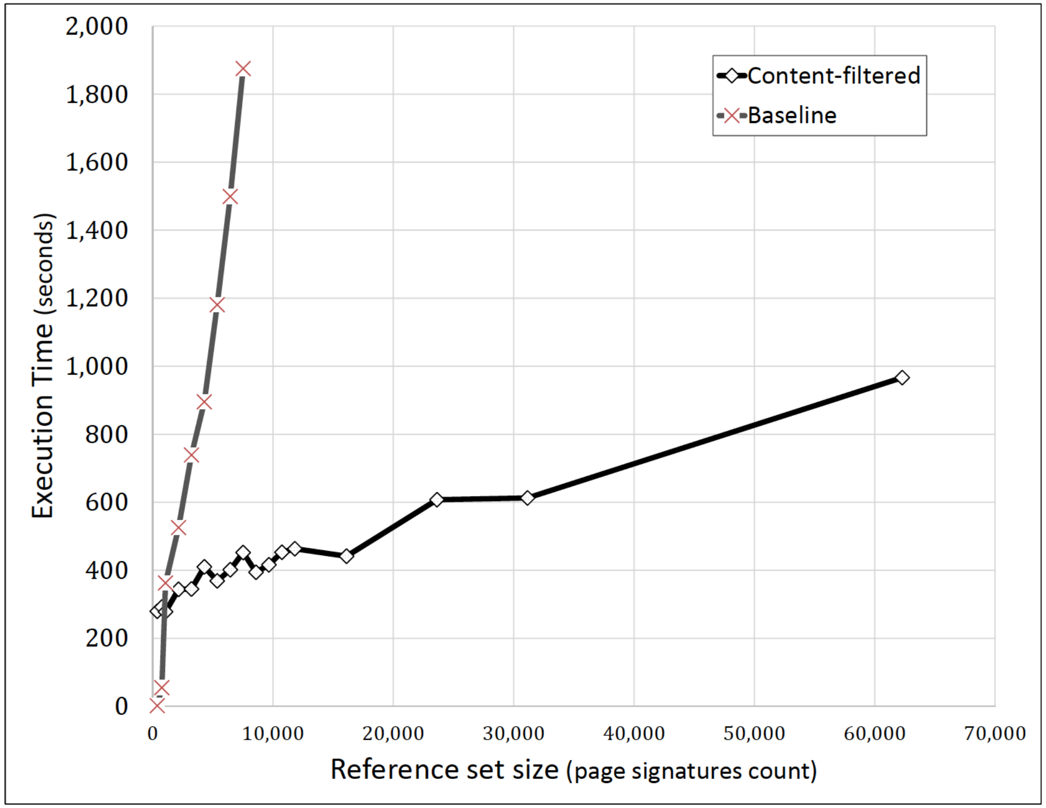
Algorithm 4 Creating a filter table.

$filter = new$ **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

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 $result$

Throughput



- PoC implementation ran on 2.6GHz Intel Core i7 CPU using a 2GB target
  - 33,554,432 memory pages to scan
- Cross-over point of algorithms is around 1000 signatures
- Further improvement: filtering out of memory pages based on content or location, sampling of signatures, and concurrent processing

Comparison with prior work

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	×	×	✓	✓

- OS-Sommelier— best representation of prior state of the art
  - Better accuracy than the approaches based on CPU registers and IDT content
- It is inherently fragile and has high sensitivity to the hypervisor
  - Dependence on specific byte patterns to identify data structures

Conclusion

- Fully automated signature generation
- Page-level accuracy of 99.79%
- Zero false negative rate
  - ensures that, if the target code is in memory, it will be found
- Perfect kernel detection (windows)
- Kernel module mapping: 97/93/100/100% for xp/vista/7 & 8
- Firefox detection – success in the worst case
- Scalable performance
- Main limiting factor in codeid is the unpredictability of paging system
  - not a notable impediment under normal workload

Q&A