How does Malware Use RDTSC? A Study on Operations Executed by Malware with CPU Cycle Measurement

Yoshihiro Oyama
University of Tsukuba

Background

- Many malware programs execute operations for analysis evasion
  - Detection of hypervisors, sandboxes, and debuggers
  - Long sleeps, logic bomb, time bomb
  - Obfuscation
- Evasion techniques are constantly advancing
- Security community needs to:
  - Correctly understand latest techniques
  - Develop effective countermeasure

Target of This Work

- Evasion operations by Windows/x86 malware
- Detection of VMs, sandboxes, or debuggers
  - Time-based
    - Taking long time for certain operation → detected
  - Using RDTSC instruction
    - Returning TSC (time stamp counter)
      - Widely used as highest-resolution clock
    - Available on x86 CPUs
    - Actually executed by many malware programs
    - Essential in microarchitecture attacks such as Meltdown and Spectre

```
t1 = RDTSC();
operation();
t2 = RDTSC();
if (t2 - t1 > thresh) {
  /* sandbox detected */
  exit(1);
}
```

Problems

- Actual RDTSC usage by malware is unclear
  - What are measured with RDTSCs?
  - Are RDTSCs often combined with CPUID?
- Intentions of such malware have not been well understood
  - Are TSCs obtained for evasion?
  - Does malware behave differently if TSCs are modified?

Goal and Method

- Goal: Clarify actual RDTSC usage by malware
  - To better understand the trends of analysis evasion using RDTSC
  - To enable future development of sophisticated countermeasure
    - E.g., automated inference of intention and choice of TSC-modifying scheme
- Method
  - Extract code fragments surrounding RDTSCs
  - Understand them
  - Develop a program that classifies them into groups
    - According to instruction sequence characteristics

Typical Code for Evasion
- Choosing Good TSC is Not Easy -

```
BOOL detect_vm()
{
  a = RDTSC();
  CPUID();
  b = RDTSC();
  return (b - a > 1000);
}
```

Determine to be inside VM if **CPUID()** takes long

Always modify TSCs to zero → Evasion prevented

```
BOOL detect_sandbox()
{
  a = RDTSC();
  SLEEP(3600); /* 1 hour */
  b = RDTSC();
  return (b - a < cpu_freq * 60 * 50);
}
```

Determine to be inside sandbox if < 50 min has passed

Always modify TSCs to zero → Sandbox detected

```
void busy_sleep(int duration)
{
  a = RDTSC();
  do {
    b = RDTSC();
  } while (b - a > duration);
}
```

Execute stealthy virtual sleep by TSC-checking busy loop

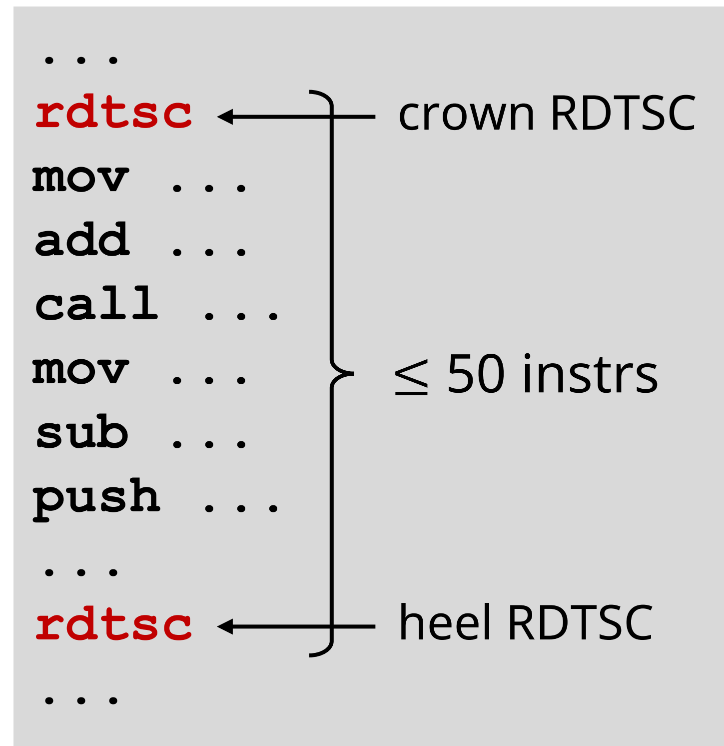
Always modify TSCs to zero → Stuck due to infinite loop

Methodology (1): Collect samples, Unpack, and Disassemble

- Download malware samples from malware-sharing website
  - 236,229 samples
  - All samples are PE32 files for Windows
  - All samples are published at the website in 2018
- Check if each sample is packed
  - Unpack if it is packed with UPX
  - Exclude samples packed with other packers
- Disassemble each sample with **`objdump`**
  - Exclude samples that cannot be disassembled

Methodology (2): Extract “RDTSC Sandwiches”

- Search for pairs of RDTSCs in a small range
- Extract code frags surrounding the pairs → RDTSC sandwich



Methodology (3): Exclude False Sandwich

- Certain ratio of disassembly results are likely to be “garbage”
  - Because of disassembling non-code such as encrypted code
  - RDTSC: 0x0f 0x31 (found from random bytes with prob. 1/65,536)

- We create heuristic rules to exclude false ones
  - E.g., Accompanied with illegal instruction

- Finally, we obtained 1,791 RDTSC sandwiches

Methodology (4): Classify Sandwiches

- We developed *the RUCS system*
  - Classifies RDTSC sandwiches into groups according to characteristics of instruction sequences
  - Implemented as a clump of pattern-matching functions
- We classified 1,791 sandwiches into 44 distinct groups

Classification Result

	Characteristic	#sandwiches	#samples	#families
1	Copying memory data	885	885	1
2	Shifting of TSC diff by 25 bits and then negating it	336	67	1
3	Measuring cycles of `Sleep()`	211	210	16
4	Measuring TSC diff between consecutive RDTSCs	74	71	10
5	TSC discarded (perhaps obfuscation)	68	68	2
6	Quadruple RDTSCs (XOR-ing `GetTickCount()` and TSC) (perhaps for random seeds)	49	49	2
7	10^n counter decrements	43	43	10
8	XOR-ing `GetTickCount()` and TSC	21	21	1
9	Quadruple RDTSCs (with PUSHA, SBB, TEST, POPA)	17	1	1
10	Function that calls `QueryPerformanceCounter()`	13	13	3
11	`timeGetTime()` loop with CPUID+RDTSC	10	10	5
12	`GetTickCount()` loop	8	8	5

Classification Result

	Characteristic	#sandwiches	#samples	#families
1	Copying memory data	885	885	1
2	Shifting of TSC diff by 25 bits and then negating it	336	67	1
3	Measuring cycles of `Sleep()`	211	210	16
4	Measuring TSC diff between consecutive RDTSCs	74	71	10
5	TSC discarded (perhaps obfuscation)	68	68	2
6	Quadruple RDTSCs (XOR-ing `GetTickCount()` and TSC) (perhaps for random seeds)	49	49	2
7	10^n counter decrements	43	43	10
8	XOR-ing `GetTickCount()` and TSC	21	21	1
9	Quadruple RDTSCs (with PUSHA, SBB, TEST, POPA)	17	1	1
10	Function that calls `QueryPerformanceCounter()`	13	13	3
11	`timeGetTime()` loop with CPUID+RDTSC	10	10	5
12	`GetTickCount()` loop	8	8	5

- Most samples measure #cycles of certain operations
- The operations are diverse

Classification Result

	Characteristic	#sandwiches	#samples	#families
1	Copying memory data	885	885	1
2	Shifting of TSC diff by 25 bits and then negating it	336	67	1
3	Measuring cycles of **`Sleep()`**	211	210	16
4	Measuring TSC diff between consecutive RDTSCs	74	71	10
5	TSC discarded (perhaps obfuscation)	68	68	2
6	Quadruple RDTSCs (XOR-ing **`GetTickCount()`** and TSC) (perhaps for random seeds)	49	49	2
7	10^n counter decrements	43	43	10
8	XOR-ing **`GetTickCount()`** and TSC	21	21	1
9	Quadruple RDTSCs (with PUSHA, SBB, TEST, POPA)	17	1	1
10	Function that calls **`QueryPerformanceCounter()`**	13	13	3
11	**`timeGetTime()`** loop with CPUID+RDTSC	10	10	5
12	**`GetTickCount()`** loop	8	8	5

Non-negligible samples execute RDTSCs for mysterious purposes

Classification Result

	Characteristic	#sandwiches	#samples	#families
1	Copying memory data	885	885	1
2	Shifting of TSC diff by 25 bits and then negating it	336	67	1
3	Measuring cycles of **`Sleep()`**	211	210	16
4	Measuring TSC diff between consecutive RDTSCs	74	71	10
5	TSC discarded (perhaps obfuscation)	68	68	2
6	Quadruple RDTSCs (XOR-ing **`GetTickCount()`** and TSC) (perhaps for random seeds)	49	49	2
7	10^n counter decrements	43	43	10
8	XOR-ing **`GetTickCount()`** and TSC	21	21	1
9	Quadruple RDTSCs (with PUSHA, SBB, TEST, POPA)	17	1	1
10	Function that calls **`QueryPerformanceCounter()`**	13	13	3
11	**`timeGetTime()`** loop with CPUID+RDTSC	10	10	5
12	**`GetTickCount()`** loop	8	8	5

CPUID-accompanying RDTSC sandwiches are minority

Characteristics Behavior (1)

Measure cycles consumed during sleep

```
rdtsc
mov [ebp+var_4], eax
mov [ebp+var_8], edx

push 1F4h ; 500
call Sleep

rdtsc
sub eax, [ebp+var_4]
sbb edx, [ebp+var_8]
```

- Lines 1–3 (rdtsc, mov, mov): Obtain TSC1 and save it
- Lines 4–5 (push, call Sleep): Sleep 500 ms
- Lines 6–8 (rdtsc, sub, sbb): Obtain TSC2 and calculate diff

Characteristics Behavior (2)

100,000 counter decrements

```
            rdtsc

            mov ecx, 100000 ; initial value
loc_44E310:
            dec ecx
            jnz short loc_44E310

            mov ebx, eax ; move TSC1
            rdtsc
            sub eax, ebx ; TSC2 - TSC1
```

Simple loop

Calculate TSC diff

Characteristics Behavior (3)

TSC as a random seed?

```
call esi ; GetTickCount
mov [esp+14h+var_10], eax
rdtsc
xor eax, edx
xor [esp+14h+var_10], eax
```

XOR-ing

1. `GetTickCount()`
2. hi32 of TSC
3. lo32 of TSC

Calculating meaningless value

```
call esi ; GetTickCount
mov [esp+14h+var_C], eax
rdtsc
xor eax, edx
xor [esp+14h+var_C], eax
...
```

Same as above

Characteristics Behavior (4)

RDTSC as NOP for obfuscation

```
            rdtsc
            nop
            mov eax, eax
loc_463896:
            rdtsc
            sub eax, eax

            ja short loc_463896

            xchg edx, edx
            mov esi, esi
            mov esi, esi
            mov ebx, ebx
            nop
```

- rdtsc / nop / mov eax, eax: Obtain and discard TSC
- rdtsc / sub eax, eax: Obtain and discard TSC
- ja short loc_463896: Never-taken branch
- xchg edx, edx … nop: Moving values between same registers

Likely for obfuscation

Characteristics Behavior (5)

CPUID to prevent out-of-order execution

```
        xor eax, eax
        xor ebx, ebx
        xor ecx, ecx
        xor edx, edx
        cpuid
        rdtsc
        mov [ebp+var_8], eax

loc_402A95:
        call edi ; timeGetTime
        sub eax, esi
        cmp eax, 3E8h
        jle short loc_402A95

        xor eax, eax
        xor ebx, ebx
        xor ecx, ecx
        xor edx, edx
        cpuid
        rdtsc
        mov [ebp+var_4], eax

        mov edx, [ebp+var_8]
        mov ecx, [ebp+var_4]
        sub ecx, edx
```

- First block (xor … mov [ebp+var_8], eax): Obtain TSC (in a better way)
- loc_402A95 block: `timeGetTime()` loop to wait
- Second block (xor … mov [ebp+var_4], eax): Obtain TSC (in a better way)
- Final block (mov edx … sub ecx, edx): Calculate TSC diff

Experiments

- Executed some samples on Cuckoo Sandbox and collected API call info
  - 99 samples (randomly-chosen one from each family in each rank)
  - Win 7 SP1 on Cuckoo 2.0.5 on Ubuntu 18.04.1
  - 120 s timeout

- Patched RDTSCs and measured the changes in API calls

- Purpose:
  - To estimate the ratio of samples affected by patches
  - To estimate the relationships between RDTSC characteristics, patch types, and degrees of behavior change

Patching

- Overwrite crown and heel of RDTSC sandwiches
  - RDTSC: 0x0f 0x31
  - Patch 1: Provide always-zero TSC
    - RDTSC (crown) → xor %eax, %eax (0x33 0xc0)
    - RDTSC (heel) → xor %eax, %eax (0x33 0xc0)
  - Patch 2: Provide small TSC diff
    - RDTSC (crown) → mov %esp, %eax (0x89 0xe0)
    - RDTSC (heel) → mov %ebp, %eax (0x89 0xe8)
  - Patch 3: Provide large TSC diff
    - RDTSC (crown) → xor %eax, %eax (0x33 0xc0)
    - RDTSC (heel) → xor %esp, %eax (0x89 0xe0)

Results from Macroscopic View

Condition	#samples
2.00 ≤ max call length ratio	2
1.50 ≤ max call length ratio < 2.00	2
1.10 ≤ max call length ratio < 1.50	4
1.05 ≤ max call length ratio < 1.10	1
0.95 ≤ max call length ratio < 1.05	74
0.90 ≤ max call length ratio < 0.95	2
0.67 ≤ max call length ratio < 0.90	6
0.50 ≤ max call length ratio < 0.67	2
0.00 ≤ max call length ratio < 0.50	6
Invoked at least one API call	99

Results from Macroscopic View

Condition	#samples
2.00 ≤ max call length ratio	2
1.50 ≤ max call length ratio < 2.00	2
1.10 ≤ max call length ratio < 1.50	4
1.05 ≤ max call length ratio < 1.10	1
0.95 ≤ max call length ratio < 1.05	74
0.90 ≤ max call length ratio < 0.95	2
0.67 ≤ max call length ratio < 0.90	6
0.50 ≤ max call length ratio < 0.67	2
0.00 ≤ max call length ratio < 0.50	6
Invoked at least one API call	99

74 out of 99:
Little affected

Results from Macroscopic View

Condition	#samples
2.00 ≤ max call length ratio	2
1.50 ≤ max call length ratio < 2.00	2
1.10 ≤ max call length ratio < 1.50	4
1.05 ≤ max call length ratio < 1.10	1
0.95 ≤ max call length ratio < 1.05	74
0.90 ≤ max call length ratio < 0.95	2
0.67 ≤ max call length ratio < 0.90	6
0.50 ≤ max call length ratio < 0.67	2
0.00 ≤ max call length ratio < 0.50	6
Invoked at least one API call	99

2 out of 99:
Length more than doubled

Results from Macroscopic View

Condition	#samples
2.00 ≤ max call length ratio	2
1.50 ≤ max call length ratio < 2.00	2
1.10 ≤ max call length ratio < 1.50	4
1.05 ≤ max call length ratio < 1.10	1
0.95 ≤ max call length ratio < 1.05	74
0.90 ≤ max call length ratio < 0.95	2
0.67 ≤ max call length ratio < 0.90	6
0.50 ≤ max call length ratio < 0.67	2
0.00 ≤ max call length ratio < 0.50	6
Invoked at least one API call	99

6 out of 99:
Length less than half

Results from Macroscopic View

Condition	#samples
2.00 ≤ max call length ratio	2
1.50 ≤ max call length ratio < 2.00	2
1.10 ≤ max call length ratio < 1.50	4
1.05 ≤ max call length ratio < 1.10	1
0.95 ≤ max call length ratio < 1.05	74
0.90 ≤ max call length ratio < 0.95	2
0.67 ≤ max call length ratio < 0.90	6
0.50 ≤ max call length ratio < 0.67	2
0.00 ≤ max call length ratio < 0.50	6
Invoked at least one API call	99

2+6 out of 99:
Greatly affected

Further investigated

Results from Microscopic View

API call length

ID	Original	Patch 1 (always-zero TSC)	Patch 2 (small TSC diff)	Patch 3 (large TSC diff)
L1	2159	2362 (109.4%)	6599 (305.7%)	5388 (249.6%)
L2	122	248 (203.3%)	248 (203.3%)	121 (99.2%)
S3	89	31 (34.8%)	31 (34.8%)	31 (34.8%)
S4	79	15 (19.0%)	15 (19.0%)	15 (19.0%)
S5	22112	625 (2.8%)	22073 (99.8%)	625 (2.8%)
S6	56916	200 (0.4%)	200 (0.4%)	200 (0.4%)
S7	161946	161946 (100.0%)	165 (0.1%)	165(0.1%)
S8	28661	3 (0.0%)	3 (0.0%)	3 (0.0%)

Observed changes:

- Long execution → early self-termination
- No crash → crash
  - Mem-access exception
  - Divide-error exception
  - INT3 exception
- Longer repetition of `timeGetTime()`
- …

Limitation: Not Lexical but Temporal RDTSC Sandwich

```
int get_tsc(void)
{
  return RDTSC();
}

void evasion(void)
{
  t1 = get_tsc();
  CPUID();
  t2 = get_tsc();

  if (t2 - t1 > 10000) {
    exit(1); /* VM detected */
  }
}
```

Out of the scope of this study

How Can the Results be Utilized in Real World?

- App 1: Sandbox reports time-related behavior signatures
  - Existing sandboxes already report signatures of some anti-analysis behavior (VMware detection, long sleeps, …)
  - Sandbox reports signatures such as “measuring the time taken for sleep” and “measuring the CPU cycles of CPUID”
- App 2: Security system detects or labels malware samples with behavior signature
- App 3: Security system responds to time-related anti-analysis ops
  - E. g., Sandbox intelligently disguises TSCs, or simply skips entire anti-analysis ops

Related Work

- Time-based detection of VMs/sandboxes/debuggers
  - Plenty of papers, reports, and systems
  - Little is known about how malware uses RDTSC and what malware intends to achieve with RDTSC
  - Our study is the first step to clarify it
- Security systems that disguise time or cycle info
  - [Kawakoya et al., 2010], [Ning et al., 2017], [Martin et al., 2012]
  - Complementary to our study
- Symbolic execution
  - angr [Shoshitaishvili et al., 2016], Triton [Saudel et al., 2015], KLEE [Cadar et al., 2008]
  - Users need to provide initial and final states, and effectiveness in (particularly automated) malware analysis is unclear
  - They are powerful in guessing inputs to prevent evasion, but malware's intention often remains a mystery
  - Our study is the first step to understand it

Summary and Future Work

- Summary
  - Malware measured CPU cycles of diverse operations
    - Counter loops, sleeps, …
  - Malware seemed to execute RDTSC for diverse purposes
    - Obfuscation, acquisition of random values, …
  - Well-known RDTSC+CPUID method was not popular
  - Patching RDTSC sandwiches alone greatly changed behavior of several malware
- Future Work
  - Further estimate usages and purposes
    - Combine with dynamic analysis?
  - Analyze temporal RDTSC sandwiches
  - Develop intelligent countermeasures against RDTSC-using malware