At the Dawn of CET:
Hunting Valid Gadget with Big Data

SOURCE Seattle 2016

Ke Sun wildsator@gmail.com

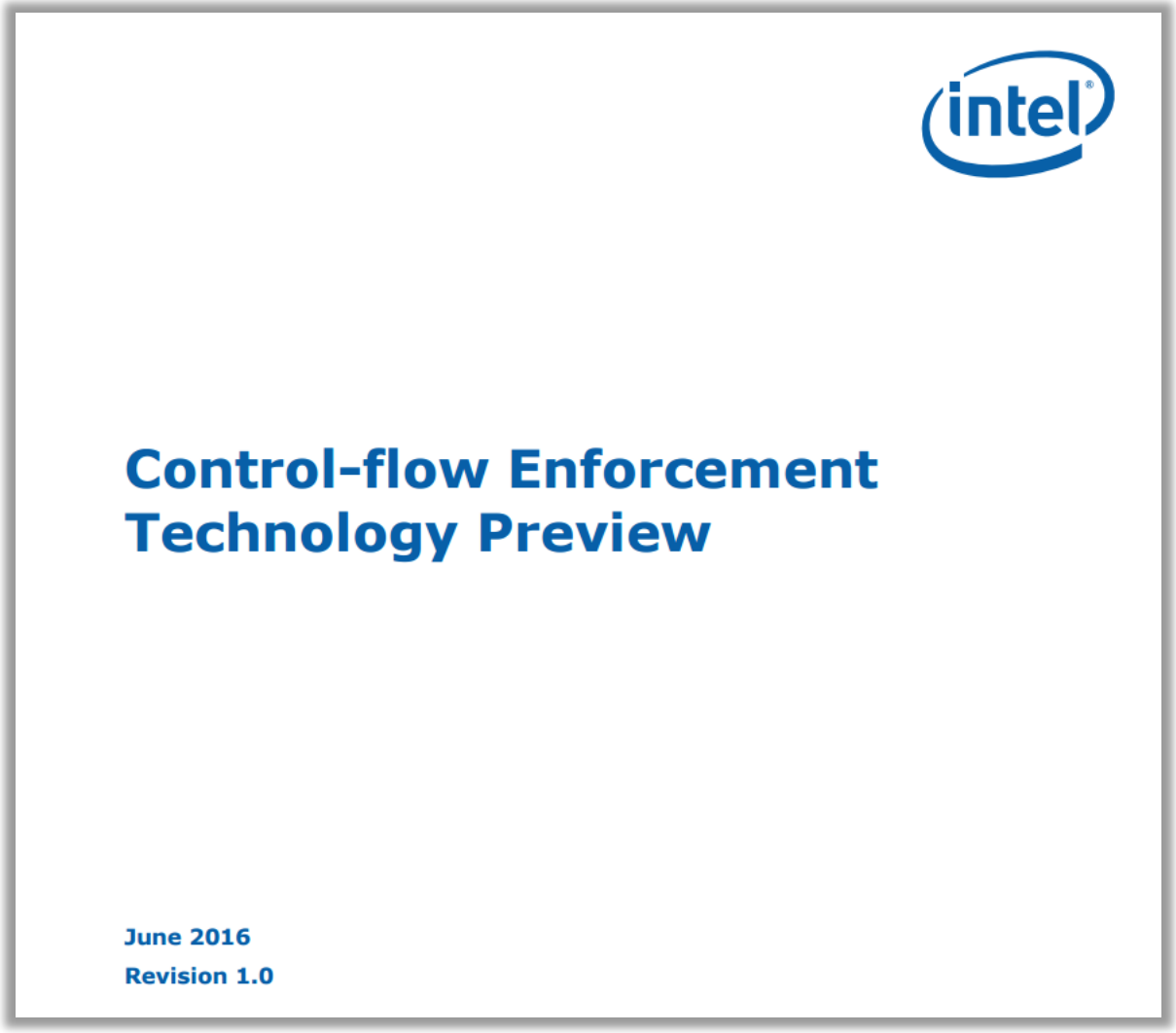
Ya Ou perfectno2015@gmail.com

Yanhui Zhao wildyz.yky@gmail.com

Xiaomin Song zosnetworking@gmail.com

Xiaoning Li ldpatchguard@gmail.com

Control-flow Enforcement Technology



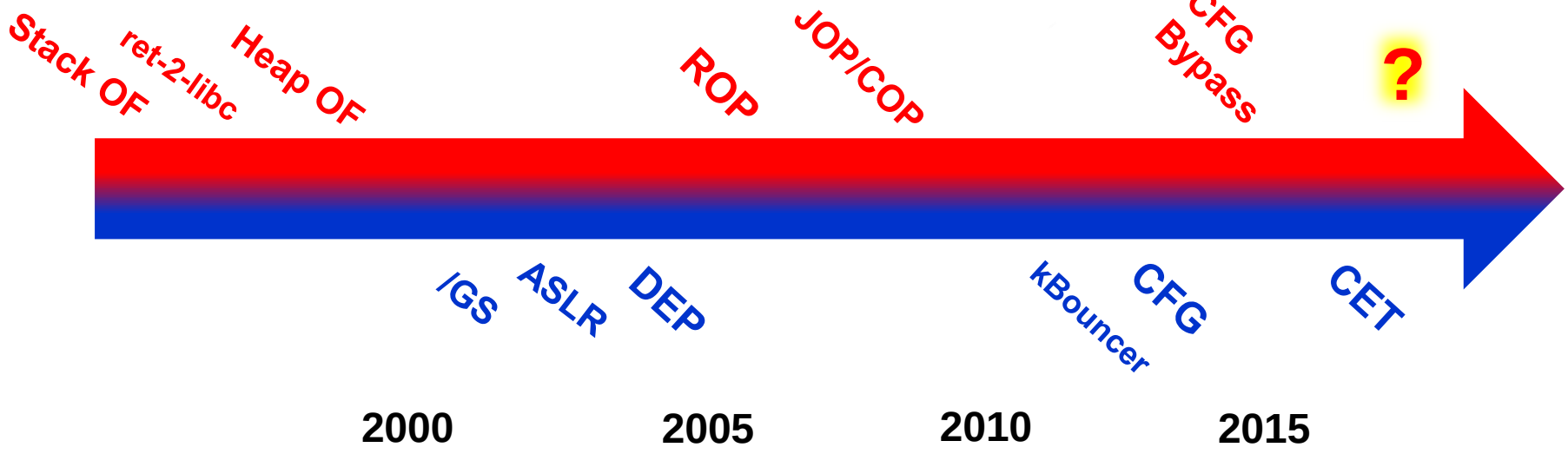
Source: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

Outline

- Mitigation Techniques before CET
- Control Flow Guard (CFG) Bypass
- Introduction to CET and CFG Bypass Mitigation by CET
- Analysis Baseline
- Gadgets Collection with PMU
- Gadgets Screening with Big Data
- Summary

History Review of Attack vs. Defense

- A history of “Spear vs. Shield”

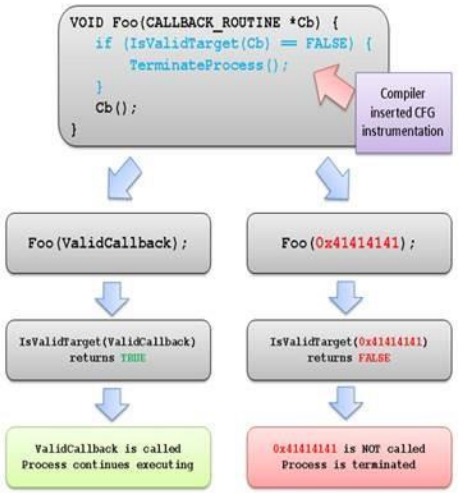


Ref: https://prezi.com/iemlmzvpnk_d/the-almost-complete-history-of-memory-corruption-attacks/

CFG Introduction

- Control Flow Guard (CFG) is a security mechanism to prevent indirect call to redirect control flow to unexpected location
  - 1st introduced in Windows 8.1, re-intro in Windows 10

- All valid function entry addresses are mapped into CFGBitmap (in average 1 bit for 8 byte) and is checked every time before an indirect call is carried out to make sure the callee's address is legal



CFG check at runtime*

offset	bit0																													bit31
+0x0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
+0x4	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
+0x8	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
+0xC	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
+0x10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+0x14	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0

...

CFG bitmap can represent the entire user mode space

*Image source: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

CFG Introduction

- Control Flow Guard is implemented by both compiler and OS

- **Compiler**
  - Insert CF check-function call (_guard_check_icall) before each indirect call
  - Generate CF function table to list all legal entry addresses (RVAs) of functions in the application
  - Specify the count of addresses in the function table
  - Label the application as CFG-enabled

- **OS**
  - Point the CF check-function pointer to ntdll!LdrpValidateUserCallTarget
  - Generate CFGBitmap based on CF function table (RVAs to runtime addresses)
  - Handle violations when CFG check fails (terminate the process by issuing an INT 29h)

Typical Control Flow Guard Bypass

- Multiple previous studies have discussed and reported CFG bypass cases and approaches.

- Possible CFG Bypass approaches:
  - Non-CFG protected modules
  - Overwrite return address
  - Unprotected indirect calls
  - Unaligned (0x10) functions

Most cases can be mitigated by CET

Overwrite Guard CF Check Function Pointer using Jscript9 CustomHeap::Heap

CORE SECURITY

March 25, 2015, By Francisco Falcón

Exploiting CVE-2015-0311, Part II: Bypassing Control Flow Guard on Windows 8.1 Update 3

Unguarded indirect call from Flash JIT compiler

Ref: https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively.pdf
Ref: https://blog.coresecurity.com/2015/03/25/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3/

CET Introduction

- Control-flow Enforcement Technology (CET) is a hardware-enforced security mechanism defend against the prevalent exploit techniques like
  - Return-oriented Programming (ROP)
  - Call/Jmp-oriented Programming (COP/JOP)
- CET contains two major parts:
  - Shadow stack, 2nd stack to double check the integrity of the return address, which detects and prevents ROP type of attack
  - Indirect branch tracking, labels all legal entries of indirect call/jmp instructions to make sure the target address valid, which hinders COP/JOP types of attack

Ref: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

CET - Shadow Stack

- Shadow stack: 2nd hidden stack which only stores return addresses
  - Protected at page level, can not be accessed by ins other than call / ret
  - Call will push return address to both normal stack and shadow stack
  - Ret will pop return addresses from both stacks and checked by CPU, if not match => control protection exception (#CP)

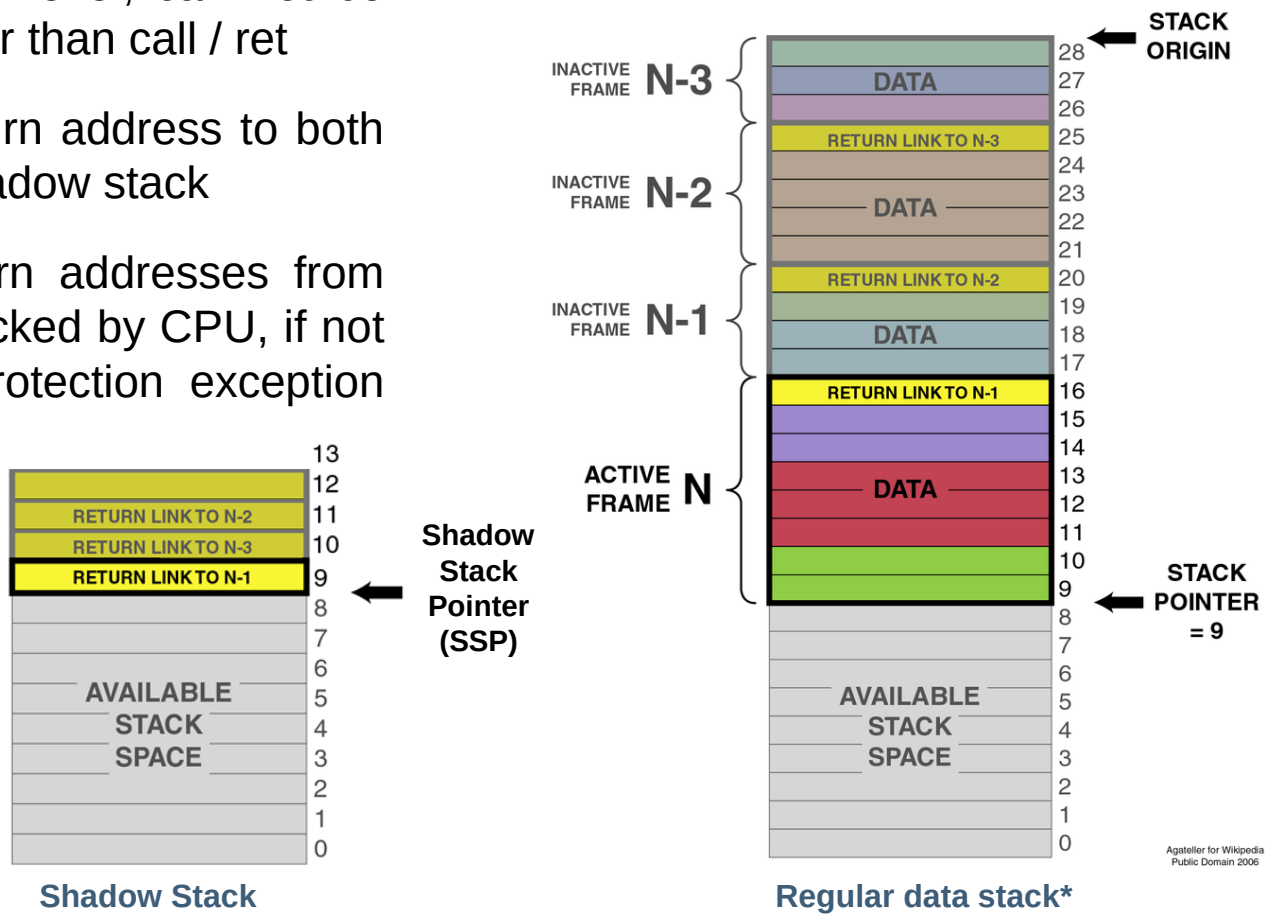


*Image source: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

CET - Indirect Branch Tracking

- Indirect branch tracking is implemented by adding a new instruction, ENDBRANCH, to mark the legal destinations of indirect branches

ENDBR32 — Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode	Instruction	Op/ En	64-Bi t Mode	Compat/ Leg Mode	Description
F3 0F 1E FB	ENDBR32	NP	Valid	Valid	Terminate indirect branch in 32 bit and compatibility mode

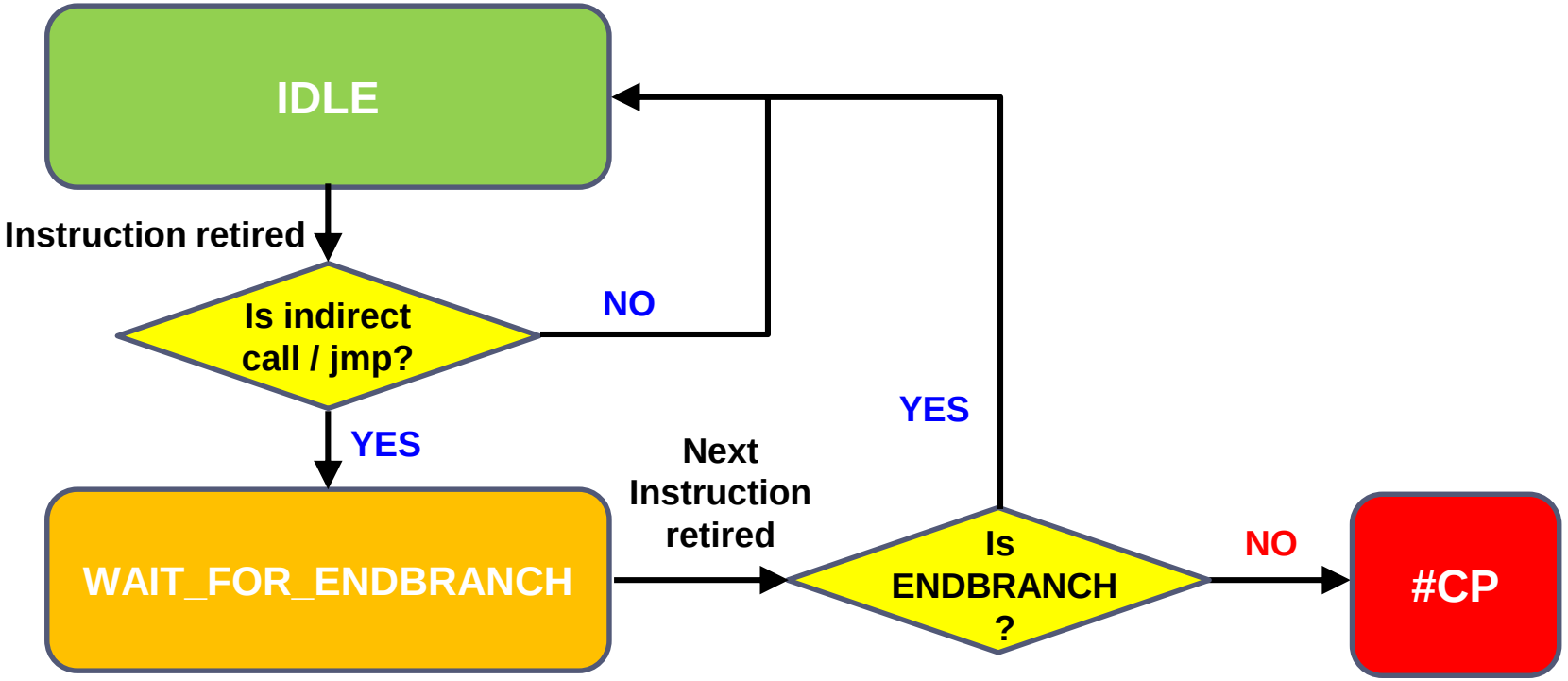
ENDBR64 — Terminate an Indirect Branch in 64-bit Mode

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E FA	ENDBR64	NP	Valid	Valid	Terminate indirect branch in 64 bit mode

Source: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

CET - Indirect Branch Tracking

- CPU implemented a two-state state machine to validate target address for indirect branches:



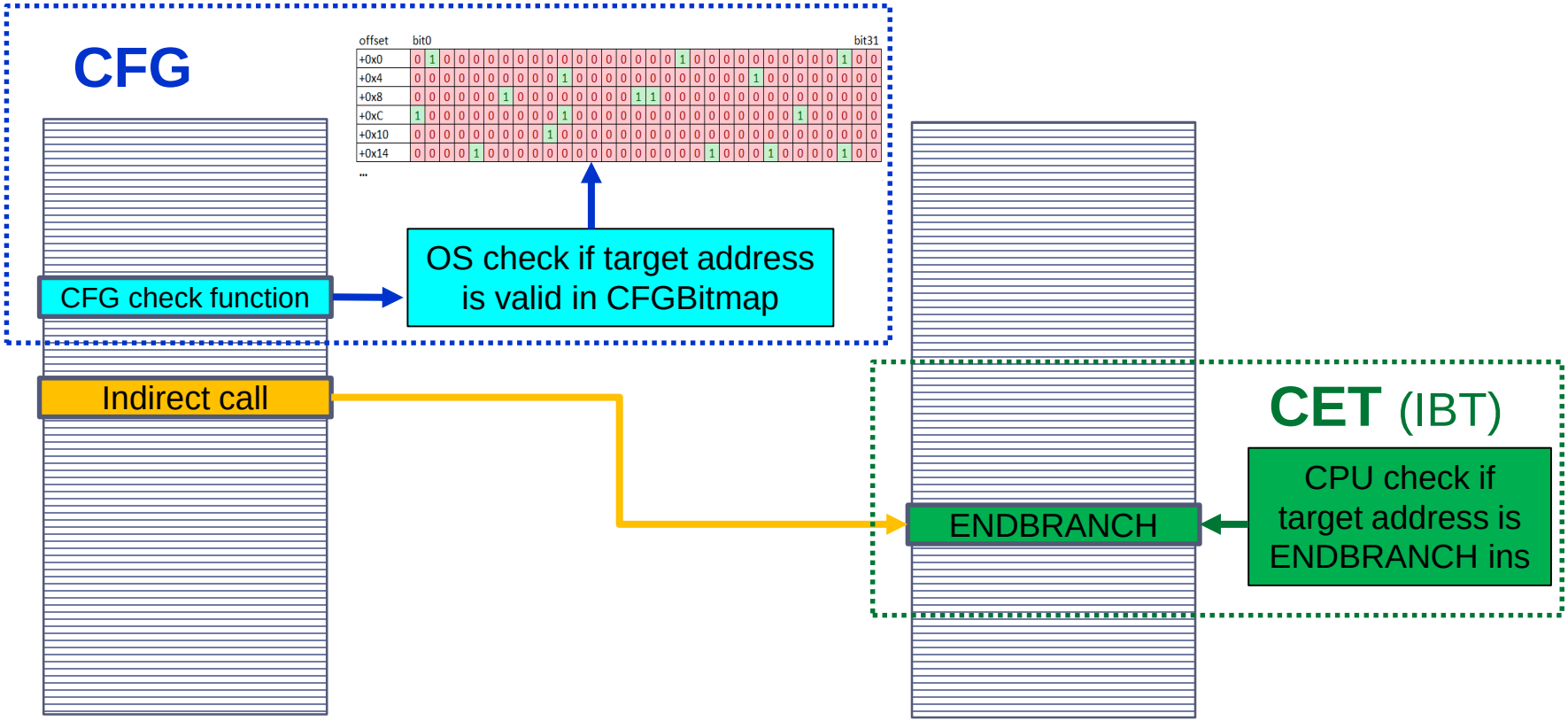
Ref: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

CFG Bypass Mitigation by CET (In most cases)

- Most of the reported CFG bypass approaches can be mitigated by CET due to its implementation of shadow stack and indirect branch labeling at hardware level:
  - Overwriting return address:
    - Can be mitigated by CET- Shadow Stack
  - Other bypass validating untrusted target address:
    - Can be mitigated by CET- Indirect Branch Tracking

CFG vs. CET (Indirect Branch Tracking)

- CFG tries to validate the target address before the indirect call in software level, while CET's indirect branch tracking checks the target address label when the indirect call takes place in hardware level



Legal Gadgets under CET

- Indirect Branch Tracking prevents the use of gadgets with untrusted entry point, but can not prevent the unintended use of legal gadgets with valid label (ENDBRANCH)

- Legal Gadgets example:

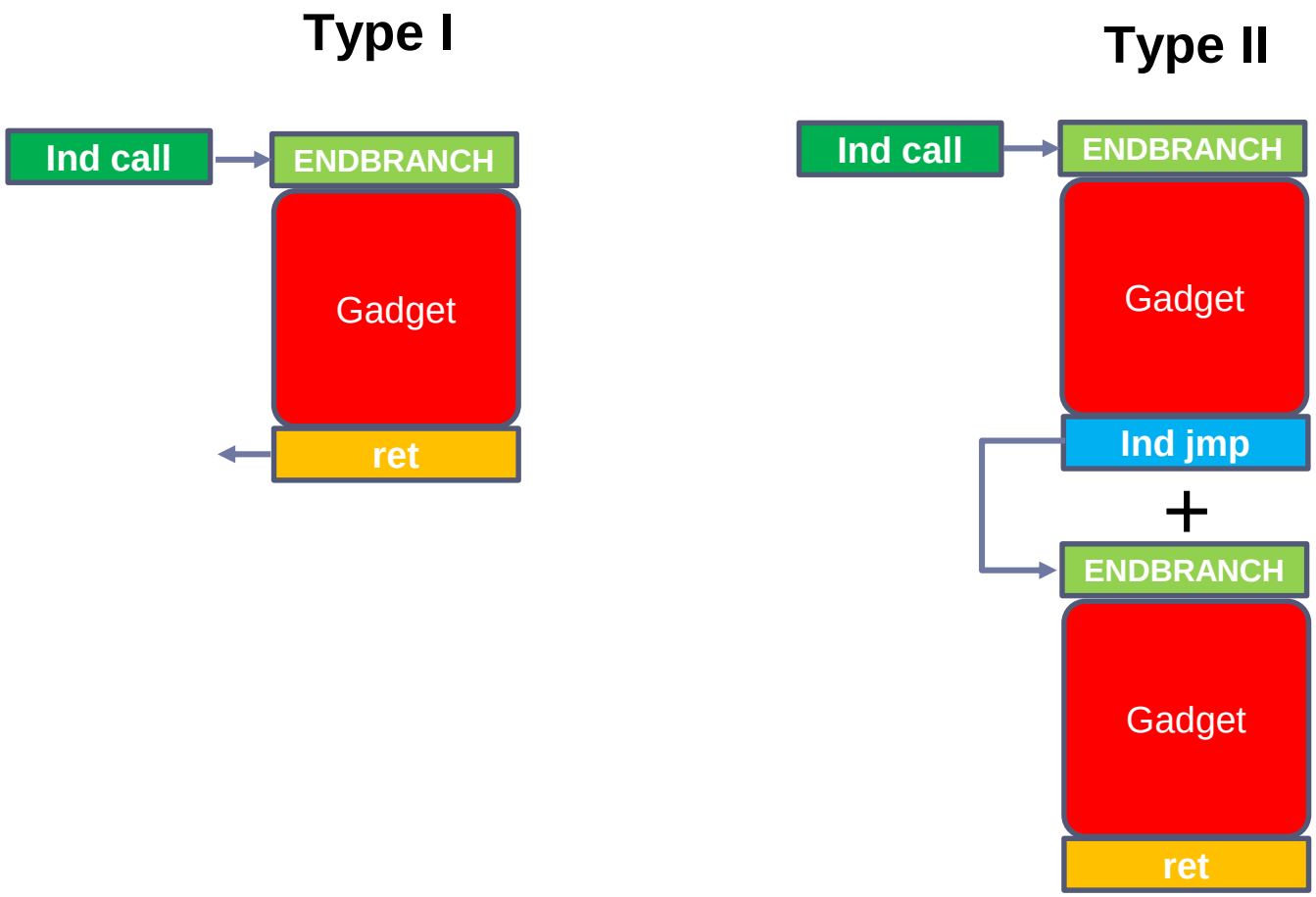
```
ENDBR32
mov edi, edi
push ebp
mov ebp, esp
mov ax, word ptr [ebp + 8];
mov word ptr [ecx + 0x20], ax;
pop ebp
ret 4
```

Analysis Baseline

- Performance Monitor Unit (PMU)-based instrumentation tool is used to track all runtime indirect branch target addresses in a non-CET-enabled system
- The assumption is all these target addresses will be legal entries under CET's indirect branch tracking (labeled with ENDBRANCH instruction)
- Code block at these target addresses are printed, disassembled and analyzed for useful gadgets
- This work only checks small gadgets with size < 32 bytes, possibly missing large gadgets

Analysis Baseline

- Typical examples of legal gadgets:

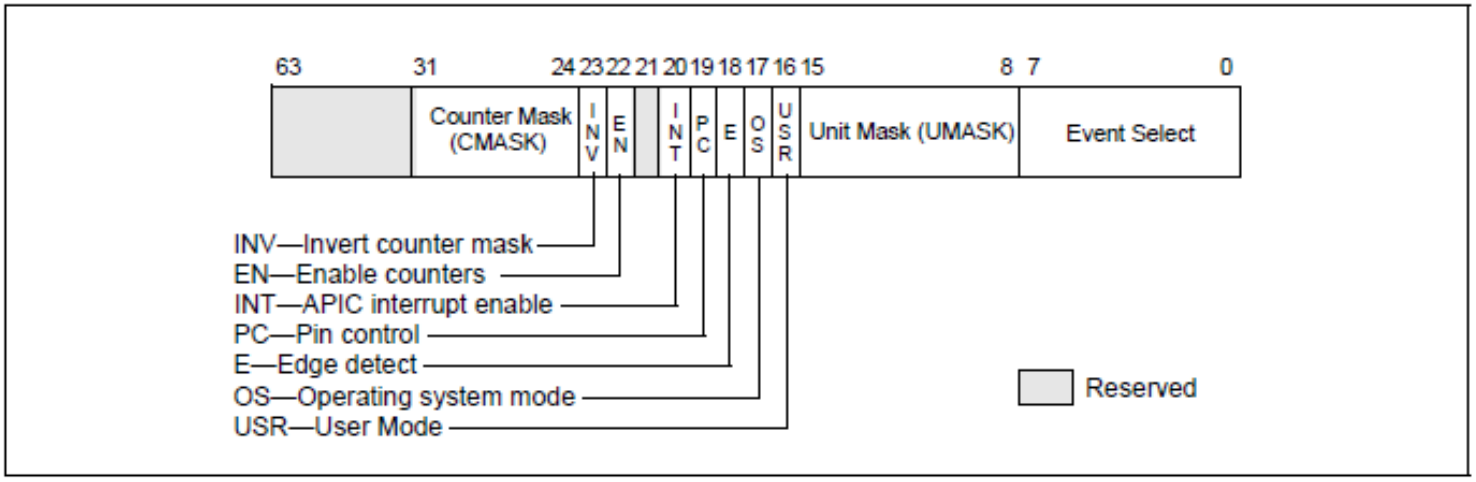


Outline

- Mitigation Techniques before CET
- Control Flow Guard (CFG) Bypass
- Introduction to CET and CFG Bypass Mitigation by CET
- Analysis Baseline
- **Gadgets Collection with PMU**
- Gadgets Screening with Big Data
- Summary

Performance Monitoring

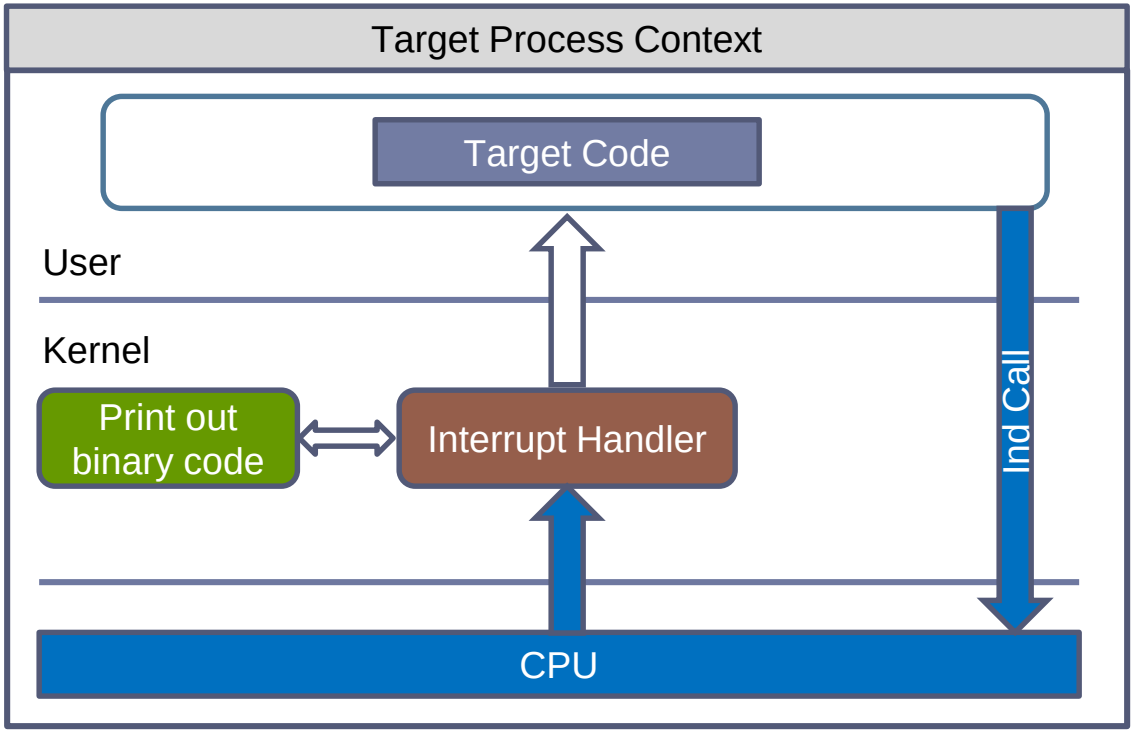
- First introduced in the Pentium processor with a set of model specific performance monitoring counter MSRs (Model-Specific Registers)
- Permit selection of processor performance parameters to be monitored and measured



IA32_PERFEVTSELx MSR

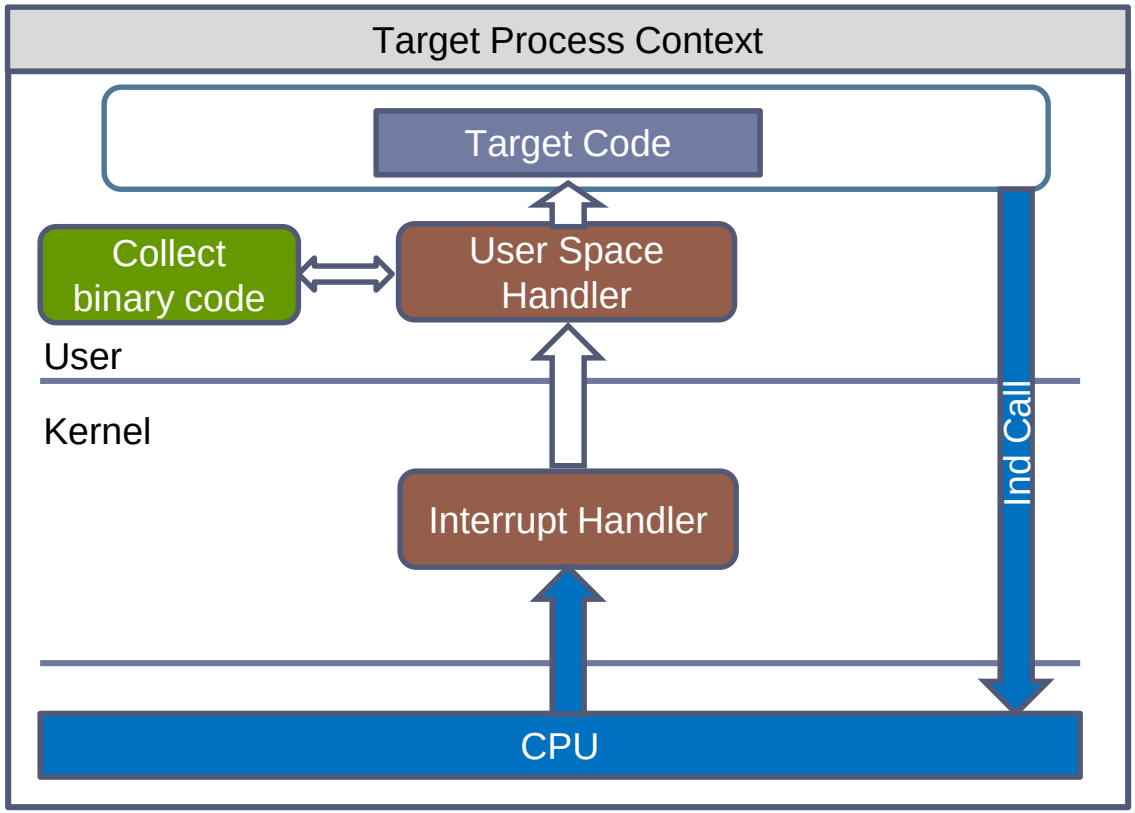
Gadget Collection with PMU

- To collect binary data after each Ind Call, we utilized PMU to track target code execution
  - Each Ind Call issues a PMI
  - Register the interrupt handler for PMI
    - 0xFE in IDT
    - Using a Windows API*
    (Ref: C. Pierce BH USA 2016)
  - Data collection
    - In Kernel Mode
    - Avoid page fault



Gadget Collection with PMU – Con'd

- Collect binary code in user mode
  - Replace interrupt EIP in Kernel stack to redirect code execution to a defined function in user mode
  - Data collection
    - In User Mode
    - Avoid dead loop



Performance Event Configuration

- CPU performance event select register (Sandy Bridge)

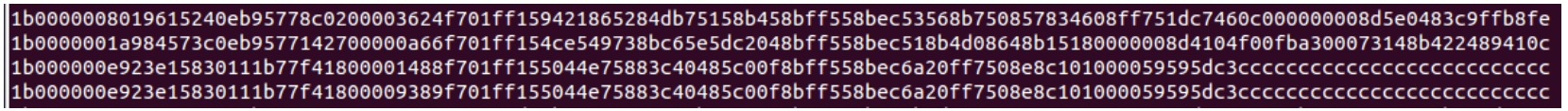
Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	84H	BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired indirect branches excluding calls and returns.	
88H	88H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN	Taken speculative and retired indirect branches that are returns.	
88H	90H	BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired direct near calls.	
88H	A0H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired indirect near calls.	

- Performance Monitor Interrupt is triggered at each indirect call instruction while running an application.
- Code stream at each legal entry of indirect call is collected for analysis.

Source: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Data Collection with PMU

- Data format



- Data volume:
  - For ie/Edge, collected data items: **69,341,184**, data file size: 4.4G.
  - For flash, collected data items: **9,949,184**, data file size: 637M.

Outline

- Mitigation Techniques before CET
- Control Flow Guard (CFG) Bypass
- Introduction to CET and CFG Bypass Mitigation by CET
- Analysis Baseline
- Gadgets Collection with PMU
- **Gadgets Screening with Big Data**
- Summary

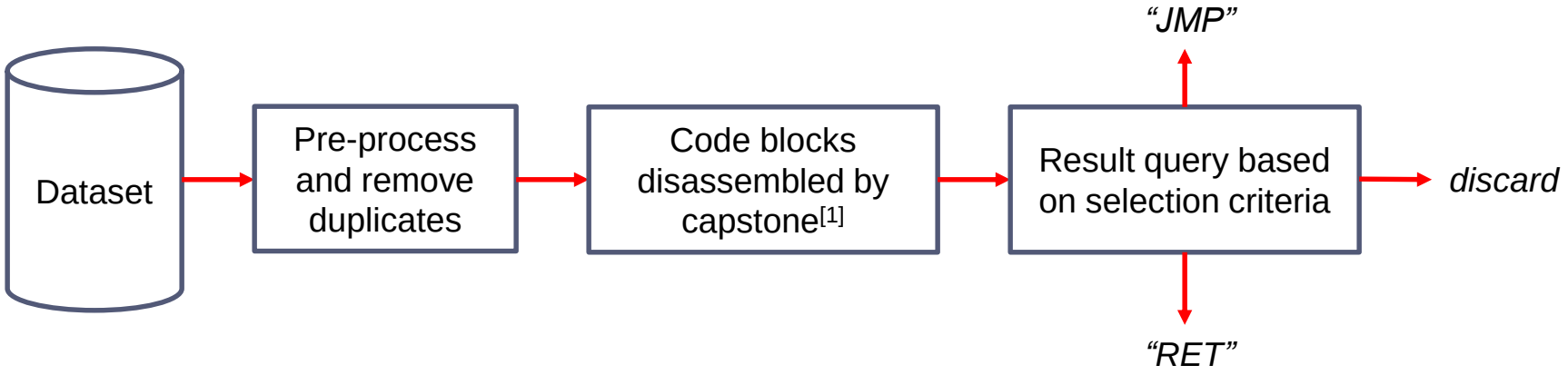
Data Analysis with Spark

- Why Spark is needed
  - *“Apache Spark is a fast and general engine for large scale data processing.”*
    - Spark uses distributed cluster computing that is easy to scale up.
    - Spark features in-memory calculation providing good performance
    - Spark program is easy to develop with high-level languages such as Scala, Java and Python
    - Spark can be easily configured to run on Big Data storing frameworks of Hadoop (focusing on storage solutions of the big data).

Source: http://spark.apache.org/

Data Analysis with Spark

- Data processing pipeline in Spark



For IE/Edge, data items reduced from **69,341,184** to **20,611**.

For flash, data items reduced from **9,949,184** to **688**

```
8b819c000000c3ccccccccccccccccc8b81a4000000c3ccccccccccccccccc
mov eax, dword ptr [ecx + 0x9c]
ret
```

√

```
8b4424048b008b8000030000e81fdcfeffffe0ccccccccccccccccccccccccc
mov eax, dword ptr [esp + 4]
mov eax, dword ptr [eax]
mov eax, dword ptr [eax + 0x300]
call 0x[illegible]6693
jmp eax
```

√

```
558bec81fca4c9151e0f8f1100000068a02a681668a4090000e83263a32c8bff
push ebp;
mov ebp, esp;
cmp esp, 0x1e15c9a4;
jg 0xe0c745;
push 0x16682aa0;
push 0x9a4;
call 0x2d842a75;
mov edi, edi;
```

X

Source: http://www.capstone-engine.org/

Valid Gadget Signature

- Unintended CET gadgets
  - Code block that contains “F3 0F 1E FA/B” and will be used as ENDBRANCH code in CET.
- Intended CET gadgets
  - (type1) Indirect call / ret

```
8b4108c3ccccccccccccccccccccccc8bff558bec8b45088941045dc20400cc
mov eax, dword ptr [ecx + 8]
ret
```

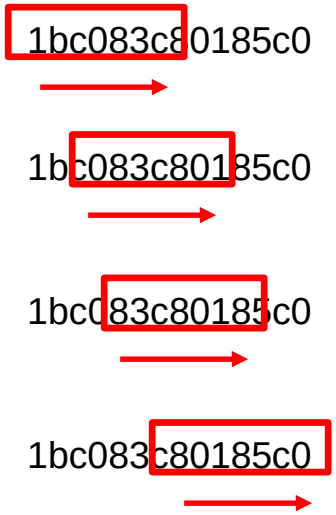
  - (type2) Indirect call / indirect jmp / ret

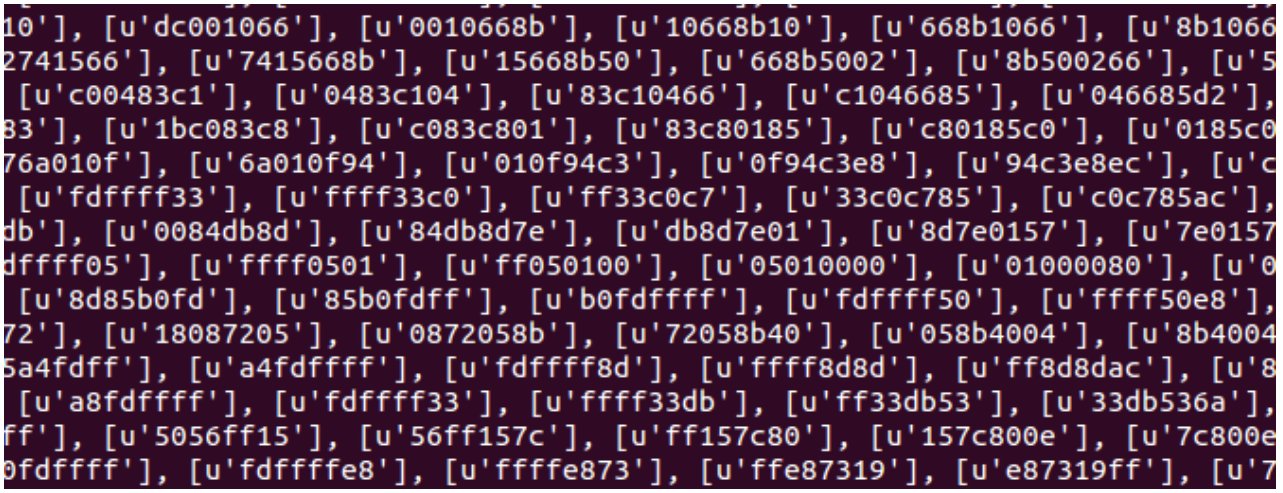
```
558bec8b813c2200008b482085c974058b015dff205dc20400ccccccccccccccc
push ebp
mov ebp, esp
mov eax, dword ptr [ecx + 0x223c]
mov ecx, dword ptr [eax + 0x20]
test ecx, ecx
je 0x108b1e86
mov eax, dword ptr [ecx]
pop ebp
jmp dword ptr [eax]
pop ebp
ret 4
```

Ref: https://software.intel.com/en-us/isa-extensions/cet-preview

Unintended Valid Gadget

- **Unintended valid gadget** is rare due to the sparsity of the binary combination of ENDBRANCH code:
  - F3 0F 1E FB
  - Use 4-byte sliding window scanned through 4056 dll files Windows 10 32-bit OS for possible match





- No hit (Good choice!)

Intended Valid Gadgets: ending with “ret”


```
b8c0276274c3ccccccccccccccccccb8bcacaa74c38bff558bec81ec100200
mov eax, 0x746227c0
ret
```

2 instructions

```
8b41140520010000c3ccccccccccccccc8bff558bec51518b4d088d45fc565750
mov eax, dword ptr [ecx + 0x14]
add eax, 0x120
ret
```

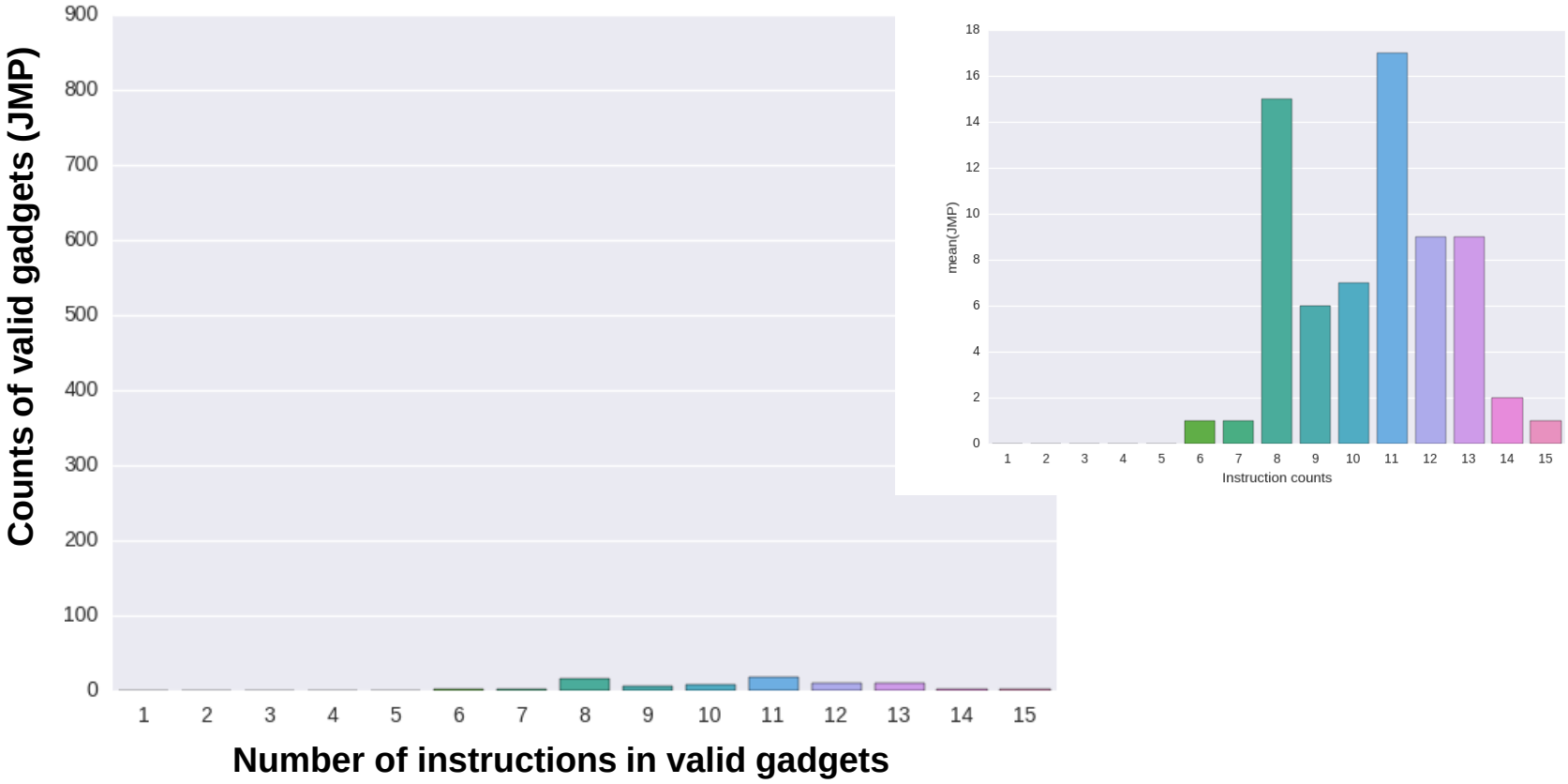
3 instructions

```
c701000000008bc1c7410400000000c3c701000000008bc1c7410400000000c7
mov dword ptr [ecx], 0
mov eax, ecx
mov dword ptr [ecx + 4], 0
ret
```

4 instructions

Dataset reduced several orders, thus can be screened and discerned by human.

Intended Valid Gadgets: containing “jmp”



Much less indirect “JMP” (tens of) captured as compared with the case of “RET” (hundreds of)

Valid Gadget Examples

- Memory access
  - Read

```
mov eax, dword ptr [ecx + 4]
ret
```

  - Write

```
mov edi, edi
push ebp
mov ebp, esp
mov ax, word ptr [ebp + 8];
mov word ptr [ecx + 0x20], ax;
pop ebp
ret 4
```

Write “1”

```
mov byte ptr [ecx + 0x4f], 1
ret
```

Write “0”

```
mov dword ptr [ecx], 0
ret
```

Valid Gadget Examples

- Arithmetic operation
  - Add

```
mov ecx, dword ptr [ecx + 0x14]
mov eax, dword ptr [ecx + 0xb0]
add eax, dword ptr [ecx + 0x88]
ret
```

  - Sub

```
mov eax, dword ptr [ecx + 0x38]
sub eax, dword ptr [ecx + 0x30]
ret
```

  - Multiply

```
mov eax, dword ptr [ecx + 0x34]
mov eax, dword ptr [eax + 0x2bc]
imul eax, dword ptr [ecx + 0x2c]
ret
```

Add 4

```
add dword ptr [ecx], 4
ret 4
```

Valid Gadget Examples

- Logic Operation
  - AND

```
mov edi, edi
push ebp
mov ebp, esp
mov edx, 0x7ff0
xor eax, eax
mov ecx, edx
and cx, word ptr [ebp + 0xe]
cmp cx, dx
setne al
pop ebp
ret
```

  - OR

```
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
not eax
or dword ptr [ecx], eax
pop ebp
ret 4
```

  - XOR

```
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
cmp eax, 0xff
jae 0x305ae529
movzx ecx, byte ptr [eax + 0x125fb740]
xor eax, ecx
pop ebp
ret
```

Valid Gadget Examples

- Stack control

```
ret 0x18
```

```
xor eax, eax
ret 0x38
```

Summary

- CET is a hardware-enforced promising security technique that can hopefully mitigate most ROP/COP/JOP type of exploitation.
- Indirect Branch Tracking of CET can prevent the use of gadgets with illegal entry point, but can not prevent the unintended use of legal gadgets
- Using PMU-based tool and Big Data analysis, legal gadgets can still be found under CET

Thank You!

Reference

- Control-flow Enforcement Technology Preview, Intel, Document Number: 334525-001, June 2016
- Control Flow Guard, https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx
- Exploring Control Flow Guard in Windows 10. Jack Tang. Trend Micro Threat Solution Team, 2015
- Windows 10 Control Flow Guard Internals. MJ0011, POC 2014
- Bypass Control Flow Guard Comprehensively, Yunhai Zhang, Blackhat 2015
- Exploiting CVE-2015-0311, Part II: Bypassing Control Flow Guard on Windows 8.1 Update 3, Francisco Falcón, Mar 2015
- Intel® 64 and IA-32 Architectures Software Developer Manuals, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. DAC 2014
- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Columbia University, 22nd USENIX Security Symposium 2013
- kBouncer: Efficient and Transparent ROP Mitigation. Vasilis Pappas. Columbia University 2012
- Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. Liwei Yuan, Weichao Xing, Haibo Chen, Binyu Zang. APSYS 2011
- Transparent Runtime Shadow Stack: Protection against malicious return address modifications. Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. 2008
- Control-Flow Integrity Principles, Implementations, and Applications. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti. CCS2005
- Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses, B Lan et al, 2015 IEEE Trustcom/BigDataSE/ISPA
- Capturing 0day Exploit With Perfectly Placed Hardware Traps, C. Pierce, M. Spisak, K. Fitch, Blackhat usa 2016
- IROP – interesting ROP gadgets, Xiaoning Li/Nicholas Carlini, Source Boston 2015
- Apache Spark, http://spark.apache.org/
- Capstone, http://www.capstone-engine.org/