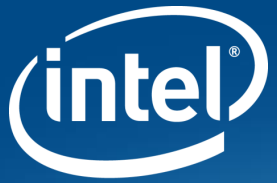


CONTROL-FLOW ENFORCEMENT TECHNOLOGY

H.J. Lu

Intel

November, 2018

Introduction

Control-flow Enforcement Technology (CET)

- An upcoming Intel® processor family feature that blocks return/jump-oriented programming (ROP) attacks
- Two components:
  - Shadow Stack (SHSTK)
  - Indirect Branch Tracking (IBT)

Control-flow Definition

The code execution path, branched by RET, JMP, or CALL.

Op Code	Operand
RET	On program stack
JMP *%rax	In memory (%rax as a pointer)
CALL *%rax	In memory (%rax as a pointer)

Shadow Stack Management

- Most of programs are compatible with SHSTK. No special treatment is needed.
- Exception handling
- Increment shadow stack pointer by the same amount.
- setjmp and longjmp
  - Save shadow stack pointer. Increment shadow stack pointer until the old value is restored.
  - ucontext

  Allocate a new shadow stack for each user provided stack with a restore token.

  Use restore token to switch shadow stack.

Indirect Branch Tracking

- All indirect branch targets must start with ENDBR64/ENDBR32.
  - ENDBR64/ENDBR32 is NOP on non-CET processors.
- The “notrack” prefix before indirect branches disables IBT.
- Optional legacy bitmap to disable IBT on legacy libraries

Mark CET-enabled Applications

- CET enabled binary must be marked.
  - A binary is marked as CET enabled only if all its components are marked as CET enabled.
- CET enabled programs are binary compatible with legacy processors.
  - CET is enabled at run-time only for CET enabled programs on CET processors.
- Linker is updated to:
  - Properly mark programs as CET enabled when all its components are marked as CET enabled.
  - Place ENDBR at all linker generated indirect branch targets.
- Kernel loader and dynamic loader are updated to properly enable CET for CET enabled programs on CET processors.

Enable CET in GCC

- Place ENDBR at all potential indirect branch targets.
- Unwind shadow stack for stack unwind intrinsics.
- Generate CET marker when CET is enabled.
- Provide a header file to generate CET marker in assembly codes.

Enable CET in Run-time Libraries

- Place ENDBR at all indirect branch targets in assembly codes.
- Mark all assembly codes as CET enabled.
- Unwind shadow stack when unwinding normal stack, including C++ exception.
  - Count number of stack frames to unwind. Increment shadow stack pointer by the same amount.
- setjmp and longjmp
  - Save shadow stack pointer. Increment shadow stack pointer until the old value is restored.
- ucontext:
  - Allocate a new shadow stack for each user provided stack with a restore token.
  - Use restore token to switch shadow stack.

Enable CET in Applications

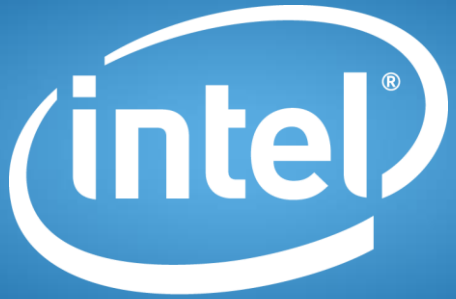
- For C/C++ sources, compile with -fcf-protection.
- For assembly sources:
  - Place ENDBR at all potential indirect branch targets in assembly codes.
  - Mark all assembly codes as CET enabled.

Status

- Specification is available by searching **“Intel CET”** or at:

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

- CET has been enabled in
  - GCC 8
  - binutils 2.31
  - glibc 2.28
- CET can be enabled in OS piece by piece
  - Start with GCC, glibc and binutils.
- Linux* kernel patches have been submitted to upstream.
  - https://github.com/yyu168/linux_cet



Software

Disclaimers

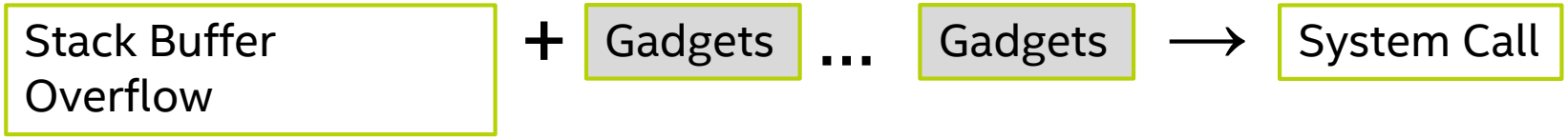
•Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

•No computer system can be absolutely secure.

•Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

•Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

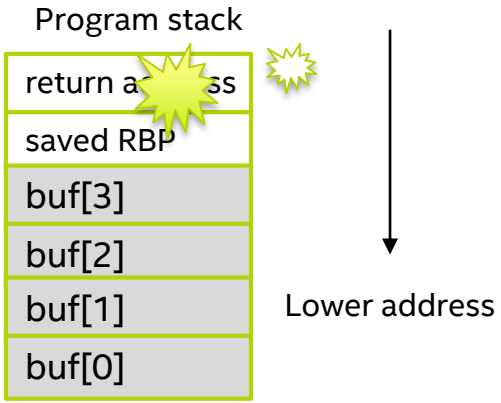
Return/Jump Oriented Programming (ROP) Attacks



No code injection is needed!

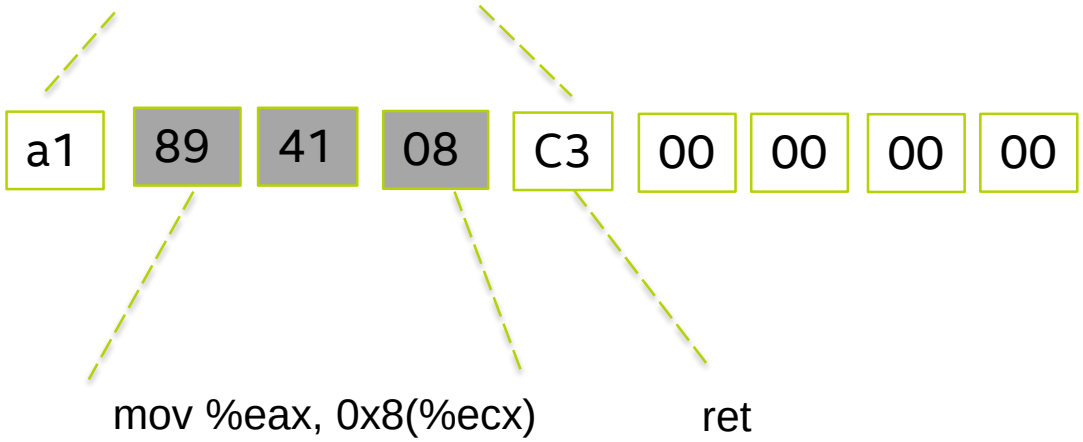
The Stack Buffer Overflow

```
void copy_string(char *input)
{
  char buf[4];
  memcpy(buf, input, strlen(input));
}
```

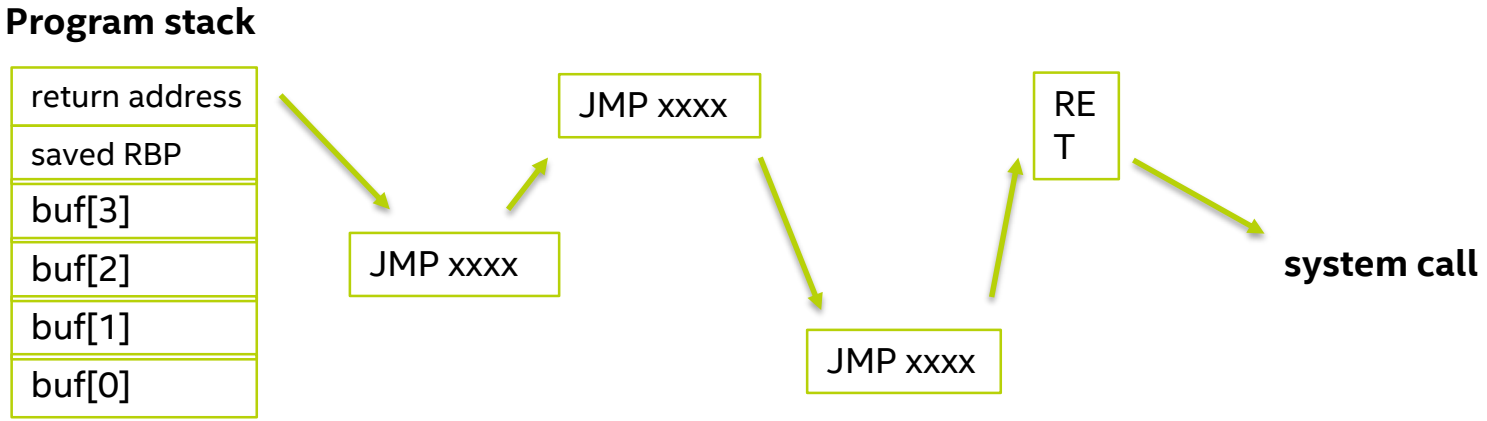


A Code Gadget Example

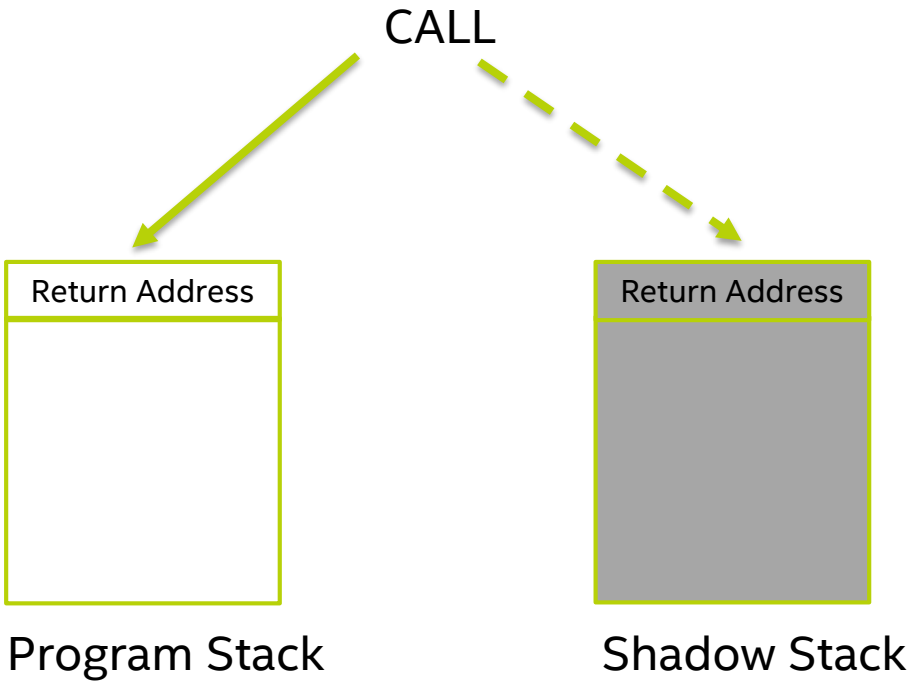
mov 0xc3084189, %eax



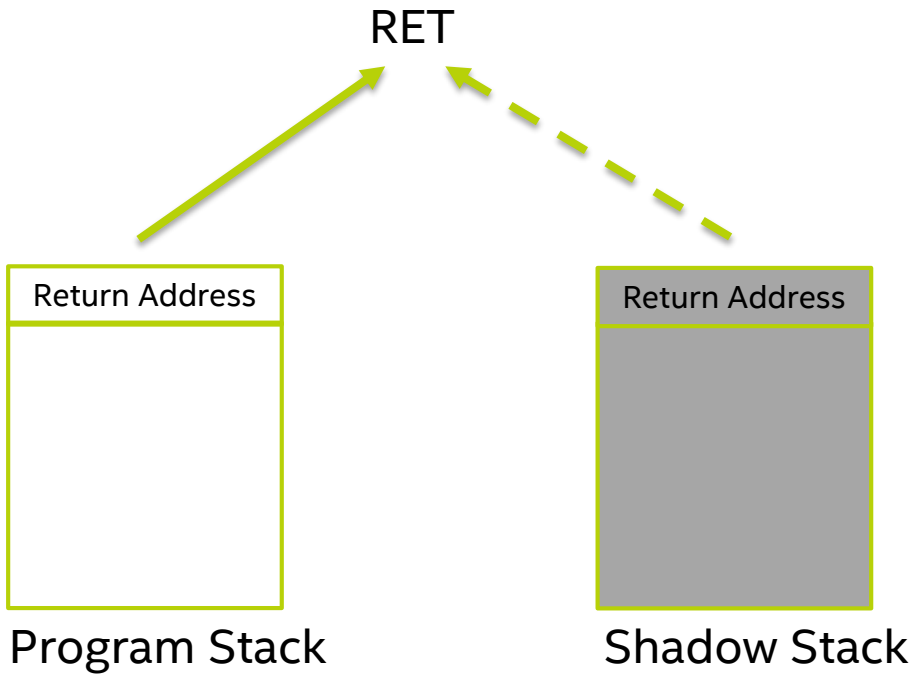
The ROP Attack



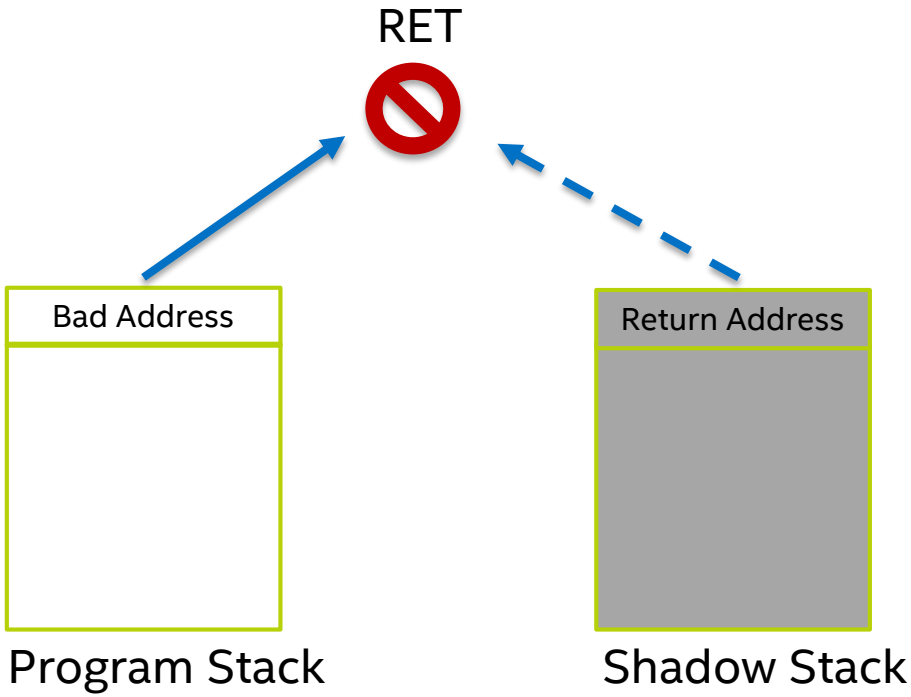
Shadow Stack Concept



Shadow Stack -- Return Address Matching



Shadow Stack Exception



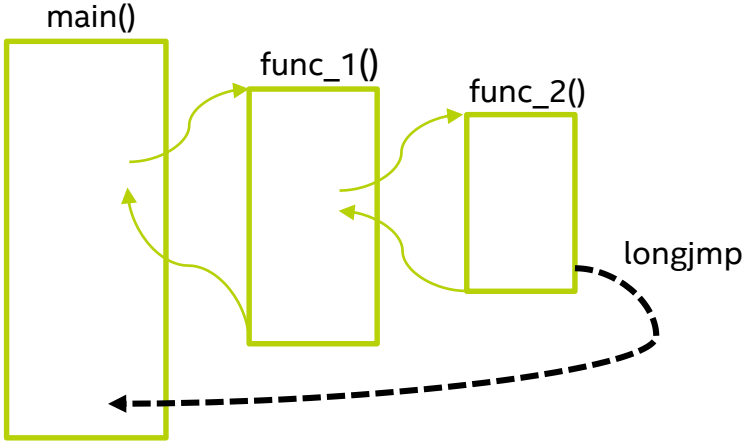
New CET Instructions

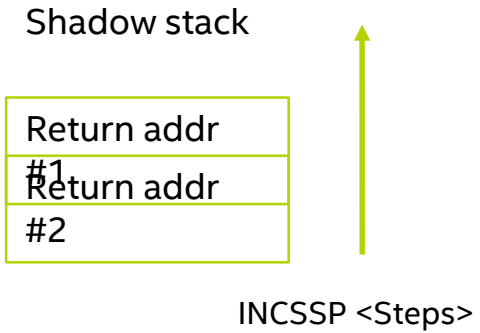
- RDSSP – Read shadow stack pointer
- INCSSP – Shadow stack unwinding

- RSTORSSP, SAVEPREVSSP – Shadow stack context switching
- SETSSBSY, CLRSSBSY – Mark shadow stack in-use

- ENDBR, No-Track – Indirect branch tracking

Shadow Stack Unwinding





Indirect Branch Tracking (IBT)

```
main() {
     int (*f)();
     f = test;
     f();
}

int test() {
     return
}
```

```
<main>:
ENDBR
:
movq    $0x4004fb, -8(%rbp)
mov     -8(%rbp), %rdx
call    *%rdx
:
retq

<test>:
ENDBR
:
add rax, rbx
:
retq
```