LINUX SECURITY SUMMIT NORTH AMERICA

# A technical deep dive into Intel CET Implementation in Linux

Jay Tharwani, NetApp Inc.

THE LINUX FOUNDATION

# What We'll Cover

01 The Problem: ROP & JOP Attacks

02 Forward-Edge Integrity: Indirect Branch Tracking (IBT)

03 The #CP Fault: Exception 21

04 Backward-Edge Defense: Shadow Stacks

05 Signal Handling & Context Switching

06 The Kernel Interface: arch_prctl(2)

07 Edge Cases & Production Deployment

---

20-minute talk + 10-minute Q&A

THE LINUX FOUNDATION

# Why Hardware CFI?

## The Problem: Code-Reuse Attacks

NX bit stopped code injection — attackers adapted

### ↩ ROP — Return-Oriented Programming

Chains small "gadgets" ending in RET
Hijacks backward-edge (return addresses on stack)

### ↪ JOP — Jump-Oriented Programming

Chains gadgets ending in JMP/CALL
Hijacks forward-edge (indirect branches)

Software mitigations (CFI, RAP) impose measurable runtime cost

## The Solution: Intel CET

### Control-flow Enforcement Technology

**Silicon-level protection**

Enforced by CPU microarchitecture — not software

**Zero runtime overhead**

Hardware checks happen in parallel with execution

**Complete coverage**

IBT protects forward-edge (JMP/CALL targets)
Shadow Stacks protect backward-edge (RET)

**Transparent to well-behaved code**

THE LINUX FOUNDATION

# Intel CET — Two Pillars

## IBT — Forward Edge

### → Indirect Branch Tracking

- Restricts valid indirect branch targets
- Only ENDBR64 instructions allowed as landing pads
- Protects against JOP attacks
- Fault: #CP with CP_ENDBR (error code 3)

**Kernel support since Linux 5.18**

GCC/Clang: -fcf-protection=branch

## Shadow Stacks — Backward Edge

### ← Hardware Return Address Stack

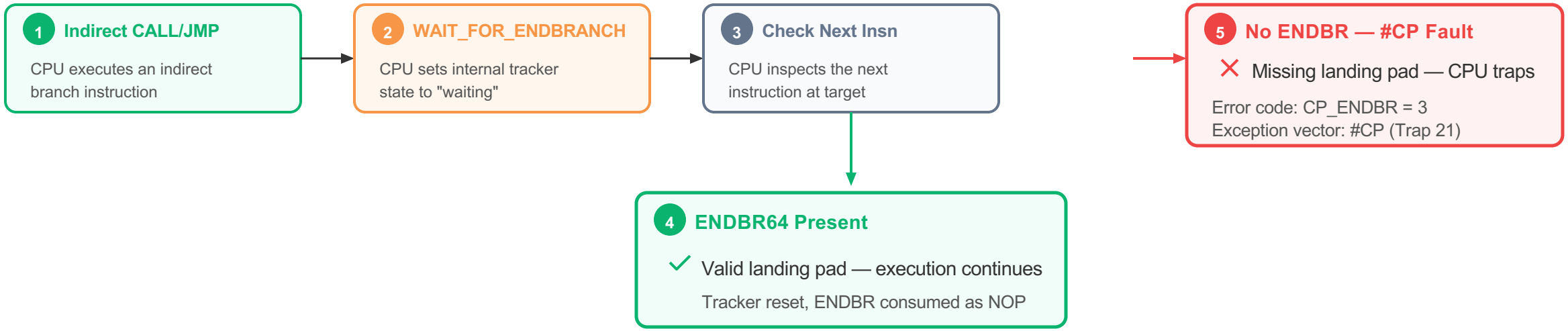
- Parallel shadow stack maintained by CPU
- CPU verifies RET address matches shadow copy
- Protects against ROP attacks
- Fault: #CP with CP_RET (error code 1)

**Userspace support since Linux 6.6**

GCC/Clang: -fcf-protection=return

**Both share exception vector: #CP (Trap 21)**

# Indirect Branch Tracking — How It Works




## Key Details

```
ENDBR64 opcode:  F3 0F 1E FA    (encodes as REP NOP on older CPUs)
ENDBR32 opcode:  F3 0F 1E FB    (32-bit variant)
```

IBT is a "legacy compatible" feature:

ENDBR decodes as NOP on hardware without CET — binaries remain backward compatible

Controlled by CR4.CET and MSR_IA32_U_CET / MSR_IA32_S_CET

THE LINUX FOUNDATION

# The ENDBR Lifecycle — Compile, Boot, Runtime

## COMPILE TIME

`gcc -fcf-protection=branch`

ENDBR at every function entry

Too many — conservative

`objtool --ibt`

Finds functions only reached by direct CALL

Records addresses in

`.ibt_endbr_seal`

**Fewer valid targets**

identified at build time

## BOOT TIME

`apply_seal_endbr()`

For each addr in `.ibt_endbr_seal:`

`is_endbr(addr)?`

uses gen_endbr() to verify

`text_poke_early()`

replaces with gen_endbr_poison()

`0xd6401f0f`

**Sealed: jump here = #CP fault**

## RUNTIME

`BPF JIT / V8 / LuaJIT`

`EMIT_ENDBR()`

= EMIT(gen_endbr(), 4)

`f3 0f 1e fa`

Writes ENDBR64 into JIT'd code so indirect calls to generated code don't trigger #CP

**JIT code = first-class IBT citizen**

eBPF tail calls land on ENDBR

THE LINUX FOUNDATION

# Generating ENDBR Without Embedding It

```
/* arch/x86/include/asm/ibt.h lines 44–57 */
#define gen_endbr()                          \

   asm (                                     \
      "mov $~0xfa1e0ff3, %[endbr]\n\t"       \
      "not %[endbr]\n\t",                    \
    : [endbr]  "=&r" (endbr));
```

**MOV loads bitwise NOT**

`~0xfa1e0ff3 = 0x05e1f00c`

**NOT flips bits back**

`NOT(0x05e1f00c) = 0xfa1e0ff3 = ENDBR64`

**Why not direct MOV?**

Direct encoding would create ENDBR gadget in .text

**Byte Transformation in Memory**

`0x05e1f00c` → NOT → `0xfa1e0ff3` → decode → `ENDBR64`

# Control Protection Exception — Error Codes

```
/* arch/x86/kernel/cet.c lines 8–18 */

enum   cp_error_code {
    CP_EC         =     BIT(15)   ,      /* SGX enclave */
    CP_RET        =     1,               /* near RET */
    CP_IRET       =     2,               /* IRET/far RET */
    CP_ENDBR      =     3,               /* missing ENDBR */
    CP_RSTORSSP   =     4,               /* RSTORSSP token */
    CP_SETSSBSY   =     5,               /* SETSSBSY token */
};
```

## Key Error Codes

| Code | Meaning | Protects |
|---|---|---|
| 1 | CP_RET<br>Shadow stack mismatch on RET | SHSTK |
| 3 | CP_ENDBR<br>Missing ENDBR at branch target | IBT |
| bit 15 | CP_EC (CP_ENCL)<br>Fault occurred inside SGX enclave | Both |

### Exception Details

- Vector 21 (#CP) — shared for all CET violations
- Error code low bits identify the specific CET sub-feature
- Bit 15 (CP_EC) is OR-ed in if fault originated from within an SGX enclave
- Kernel handler dispatches based on error code to distinct recovery paths

THE LINUX FOUNDATION

# Kernel #CP Handler — Routing Faults

```
/* arch/x86/kernel/cet.c lines 158–168 */

DEFINE_IDTENTRY_ERRORCODE  (exc_control_protection)
{
      if(user_mode(regs)) {
          if(cpu_feature_enabled(X86_FEATURE_USER_SHSTK))
              do_user_cp_fault(regs, error_code);
          else
              do_unexpected_cp(regs, error_code);
      }else{
          if(cpu_feature_enabled(X86_FEATURE_IBT))
              do_kernel_cp_fault(regs, error_code);
          else
              do_unexpected_cp(regs, error_code);
      }
}
```

## Dispatch Logic

**User-mode #CP**

do_user_cp_fault → SIGSEGV / SEGV_CPERR

Logs IP, SP, SSP, error code; kills process

**Kernel-mode #CP**

Only handles CP_ENDBR; others → unexpected

Default: BUG() | ibt=warn: WARN()

Has self-test escape for ibt_selftest_noendbr

**FRED-aware**

Clears WFE (Wait-For-ENDBRANCH) state on entry

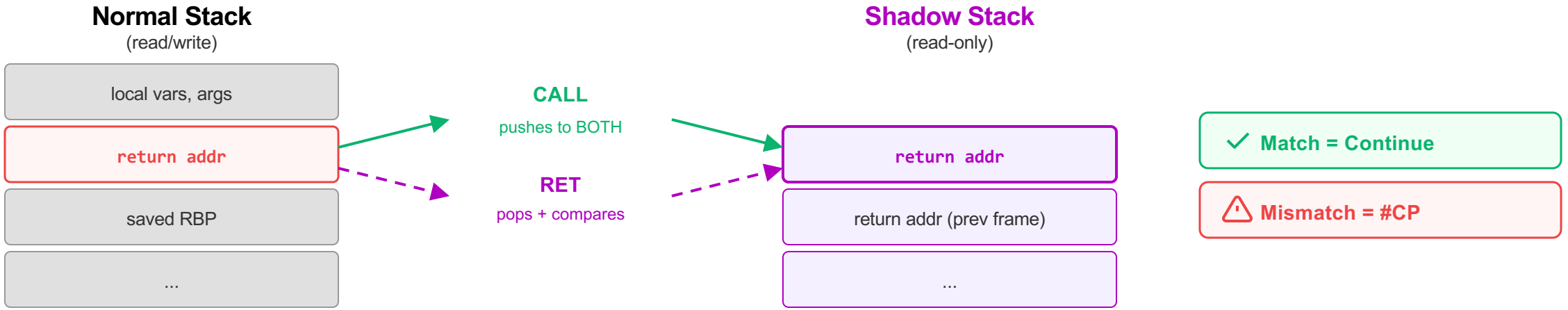
**Handler Flow Summary**

```
#CP trap → user_mode? → SIGSEGV(SEGV_CPERR) : (CP_ENDBR? → ibt_fatal? BUG : WARN)
```

FRED (Flexible Return and Event Delivery) automatically manages the tracker state

THE LINUX FOUNDATION

# Hardware Shadow Stack — The Mechanism




## Critical Properties

- Shadow stack is **NOT** writable by normal MOV/PUSH instructions
- Only **CALL**, **RET**, and special CET instructions (WRSS, INCSSP) modify it
- `MSR_IA32_PL3_SSP` holds the current user Shadow Stack Pointer
- Mismatch on RET triggers #CP with error code `CP_RET = 1`

```
MSR_IA32_PL3_SSP = 0x000006A4 MSR_IA32_U_CET = 0x000006A0
```

# Enabling Shadow Stacks — shstk_setup()

```
// arch/x86/kernel/shstk.c, lines 149-177

static int shstk_setup(void)
{
    struct thread_shstk *shstk =
        &current->thread.shstk;
    unsigned long addr, size;

    if (features_enabled(ARCH_SHSTK_SHSTK))
        return 0;  /* already enabled */

    size = adjust_shstk_size(0);
    addr = alloc_shstk(0, size, 0, false);

    fpregs_lock_and_load();
    wrmsrq(MSR_IA32_PL3_SSP, addr + size);
    wrmsrq(MSR_IA32_U_CET, CET_SHSTK_EN);
    fpregs_unlock();

    shstk->base = addr;
    shstk->size = size;
    features_set(ARCH_SHSTK_SHSTK);
    return 0;
}
```

**Size allocation**

min(RLIMIT_STACK, 4GB), page-aligned

**MAP_ABOVE4G**

Prevents 32-bit address aliasing attacks

**SSP at top**

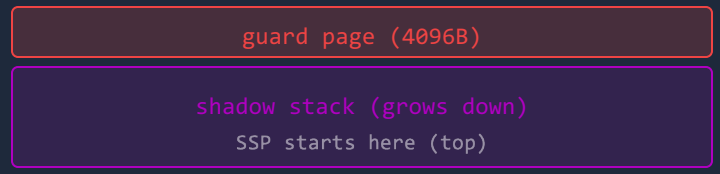
addr + size (stacks grow downward)

**Guard page protection**

INCSSP max reach = **2040 bytes**

Guard page = 4096 bytes > max reach

**Memory Layout**




THE LINUX FOUNDATION

# Signals vs. Shadow Stacks

## The Problem

1. Signal delivery diverts control flow to handler
2. Handler returns via `sa_restorer` trampoline (RET)
3. Shadow stack expects the **original caller's** return address
4. Without fixup: **#CP fault on every signal return**

## Signal Delivery Protocol (Solution)

**Step 1**
Push sigframe token
`old_ssp | BIT(63)`

→

**Step 2**
Push sa_restorer
onto shadow stack

→

**Step 3**
Update SSP
`MSR_IA32_PL3_SSP`

→

**Step 4**
Handler's RET
matches sa_restorer

→

**Step 5**
rt_sigreturn validates token
restores original SSP

**Security:** BIT(63) ensures token can never match a RET address — attacker cannot forge tokens without WRSS or kernel access

restore_signal_shadow_stack() reverses the entire protocol on sigreturn

THE LINUX FOUNDATION

# setup_signal_shadow_stack()

```
// arch/x86/kernel/shstk.c, lines 384-416

int setup_signal_shadow_stack(
    struct ksignal *ksig)
{
    void __user *restorer =
        ksig->ka.sa.sa_restorer;
    unsigned long ssp;

    ssp = get_user_shstk_addr();

    /* Push sigframe token:
     * old_ssp | BIT(63) */
    shstk_push_sigframe(&ssp);

    /* Push restorer so handler's
     * RET matches */
    ssp -= SS_FRAME_SIZE;
    write_user_shstk_64(
        (u64 __user *)ssp,
        (u64)restorer);

    fpregs_lock_and_load();
    wrmsrq(MSR_IA32_PL3_SSP, ssp);
    fpregs_unlock();

    return 0;
}
```

**SHSTK_DATA_BIT (bit 63)**

Ensures token can never match
a legitimate RET address

**sa_restorer**

`= __restore_rt`

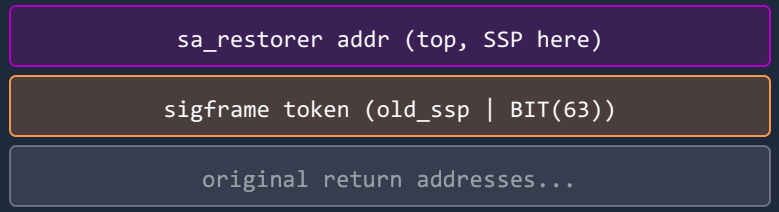
Calls rt_sigreturn syscall

**Reverse on sigreturn**

`restore_signal_shadow_stack()`

Validates token, restores original SSP

**Shadow Stack State During Signal**




THE LINUX FOUNDATION

# Shadow Stack Lifecycle — Fork, Threads, vfork

```
// arch/x86/kernel/shstk.c, lines 186-226 (simplified)

unsigned long shstk_alloc_thread_stack(
    struct task_struct *tsk,
    u64 clone_flags,
    unsigned long stack_size)
{
    if (clone_flags & CLONE_VFORK) {
        shstk->base = 0;
        /* share parent's */
        shstk->size = 0;
        return 0;
    }
    if (!(clone_flags & CLONE_VM))
        return 0; /* fork: COW */

    /* pthread_create: new stack */
    size = adjust_shstk_size(stack_size);
    addr = alloc_shstk(0, size, 0, false);
    shstk->base = addr;
    shstk->size = size;
    return addr + size;
}
```

| Scenario | Flags | Action |
|---|---|---|
| `fork()` | `!CLONE_VM` | COW copy (free) |
| `vfork()` | `CLONE_VFORK` | Share parent's base=0, size=0 |
| `pthread_create()` | `CLONE_VM` | Allocate new returns addr+size |

**Why base=0 for vfork?**

Prevents shstk_free() from unmapping the parent's shadow stack

Returned SSP is stored into child's XSAVE area by `update_fpu_shstk()`

Child resumes with correct shadow stack pointer on first context switch

# SSP During Context Switch

```
// arch/x86/kernel/fpu/core.c, lines 634-658

static int update_fpu_shstk(
    struct task_struct *dst,
    unsigned long ssp)
{
    struct cet_user_state *xstate;

    if (!ssp)
        return 0;

    xstate = get_xsave_addr(
        &x86_task_fpu(dst)
            ->fpstate->regs.xsave,
        XFEATURE_CET_USER);

    xstate->user_ssp = (u64)ssp;
    return 0;
}
```

**XFEATURE_CET_USER**

SSP stored in XSAVE area under this feature

**Context switch: automatic**

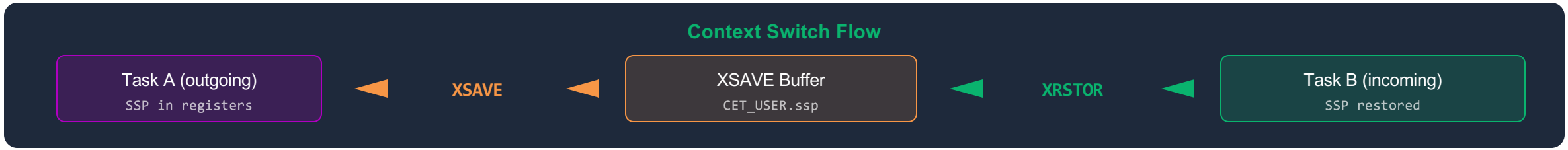
XSAVE saves outgoing, XRSTOR loads incoming SSP

**Zero scheduler overhead**

No explicit MSR read/write in fast path

**fpu_clone()**

Calls update_fpu_shstk() to set child's initial SSP



# Userspace Control — arch_prctl(2) for CET

```
// arch/x86/include/uapi/asm/prctl.h

#define ARCH_SHSTK_ENABLE   0x5001
#define ARCH_SHSTK_DISABLE  0x5002
#define ARCH_SHSTK_LOCK     0x5003
#define ARCH_SHSTK_UNLOCK   0x5004
#define ARCH_SHSTK_STATUS   0x5005

#define ARCH_SHSTK_SHSTK  (1ULL << 0)
#define ARCH_SHSTK_WRSS   (1ULL << 1)
```

```
// Usage pattern

/* Enable shadow stack */
arch_prctl(ARCH_SHSTK_ENABLE,
           ARCH_SHSTK_SHSTK);

/* Lock -- prevents disabling */
arch_prctl(ARCH_SHSTK_LOCK,
           ARCH_SHSTK_SHSTK);
```

**LOCK**

Prevents disable even by the process itself

Attacker with code execution cannot call DISABLE

**UNLOCK**

Only via ptrace (for CRIU/checkpoint-restore)

External debugger privilege required

**glibc auto-enable**

CET-compiled binaries with GNU property notes:

Dynamic linker enables + locks before main()

## Production Escape Hatch

```
GLIBC_TUNABLES=glibc.cpu.hwcaps=-SHSTK ./app
```

Per-process, no kernel restart needed — useful for legacy apps and incompatible JITs

# Edge Cases & Deployment

## setjmp / longjmp

**Problem:**
longjmp skips intermediate frames, shadow stack has stale entries

**Solution:**
glibc uses `INCSSP` to advance past stale entries

**Deep longjmp:**
May require `WRSS` (separate opt-in via arch_prctl)

## JIT Engines

**V8, LuaJIT, eBPF**
Must emit `ENDBR` at all generated indirect targets

**map_shadow_stack**
Syscall for custom coroutine and fiber shadow stacks

**eBPF JIT:**
Tail calls must land on ENDBR instruction

## GLIBC_TUNABLES

```
glibc.cpu.hwcaps=
-SHSTK
```

Disables shadow stacks at dynamic linker level

**Benefits:**
+ Per-process control
+ No kernel restart
+ Legacy app support
+ Debugging use cases

# Summary & Timeline

**Linux 5.18**
May 2022
Kernel IBT

**Linux 6.4**
Jun 2023
FineIBT

**Linux 6.6**
Oct 2023
User Shadow Stacks

**Linux 6.6+**
Ongoing
map_shadow_stack

## Key Takeaways

1. CET delivers hardware CFI with zero performance tax on the fast path
2. IBT protects forward edges — ENDBR validation, sealing, and #CP enforcement
3. Shadow Stacks guard backward edges — read-only return addresses + signal tokens
4. Production-ready: glibc integration, arch_prctl locking, GLIBC_TUNABLES
5. Entire implementation lives files under arch/x86/

Scan for slides

## Questions?

Jay Tharwani
jtharwan@alumni.uncc.edu

THE LINUX FOUNDATION