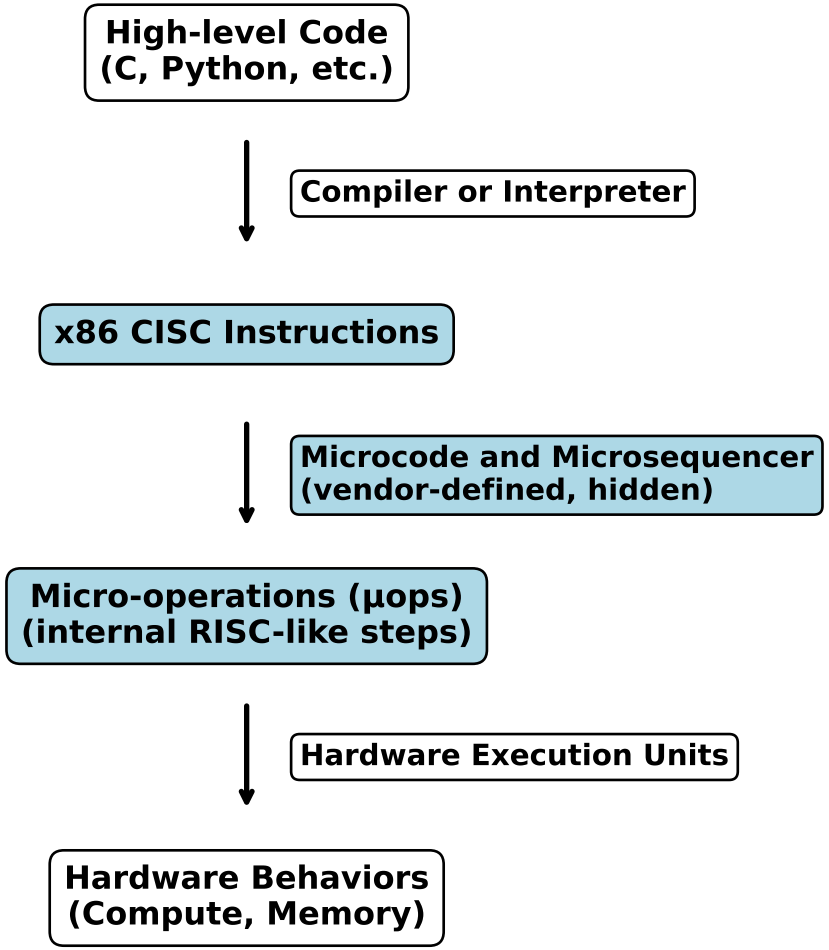
reInstruct: Toward OS-aware CPU microcode reprogramming

Yubo Wang
Penn State
yubow@psu.edu

Prof. Ruslan Nikolaev
Penn State
rnikola@psu.edu

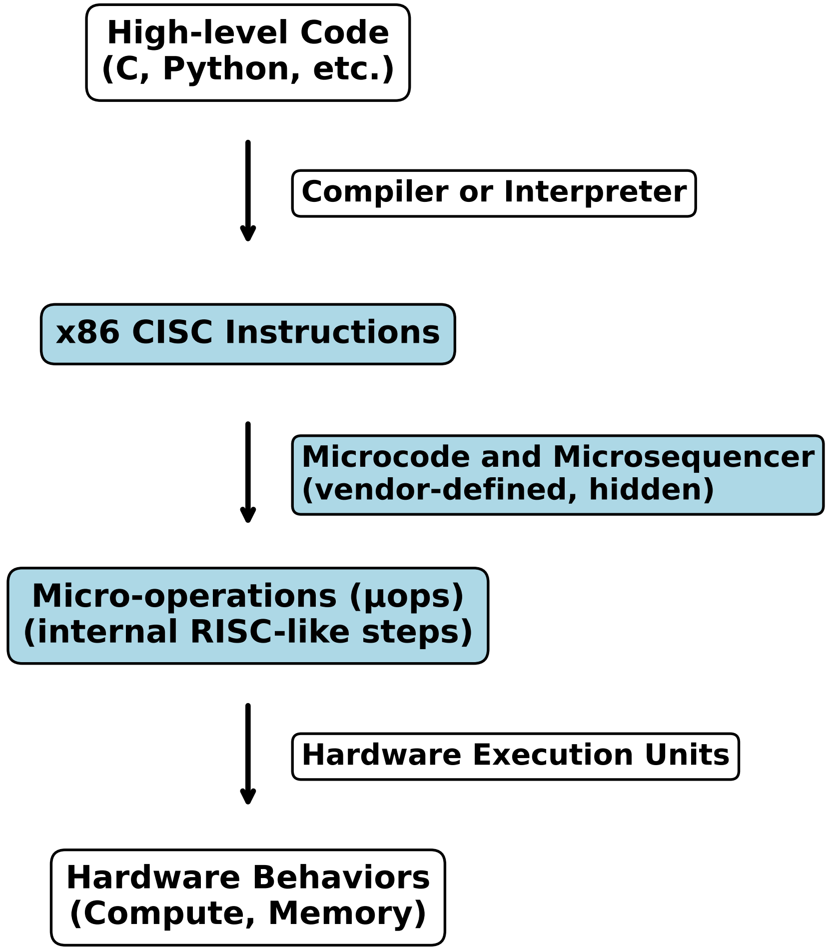
Prof. Binoy Ravindran
Virginia Tech
binoy@vt.edu

What is Microcode?



What is Microcode?

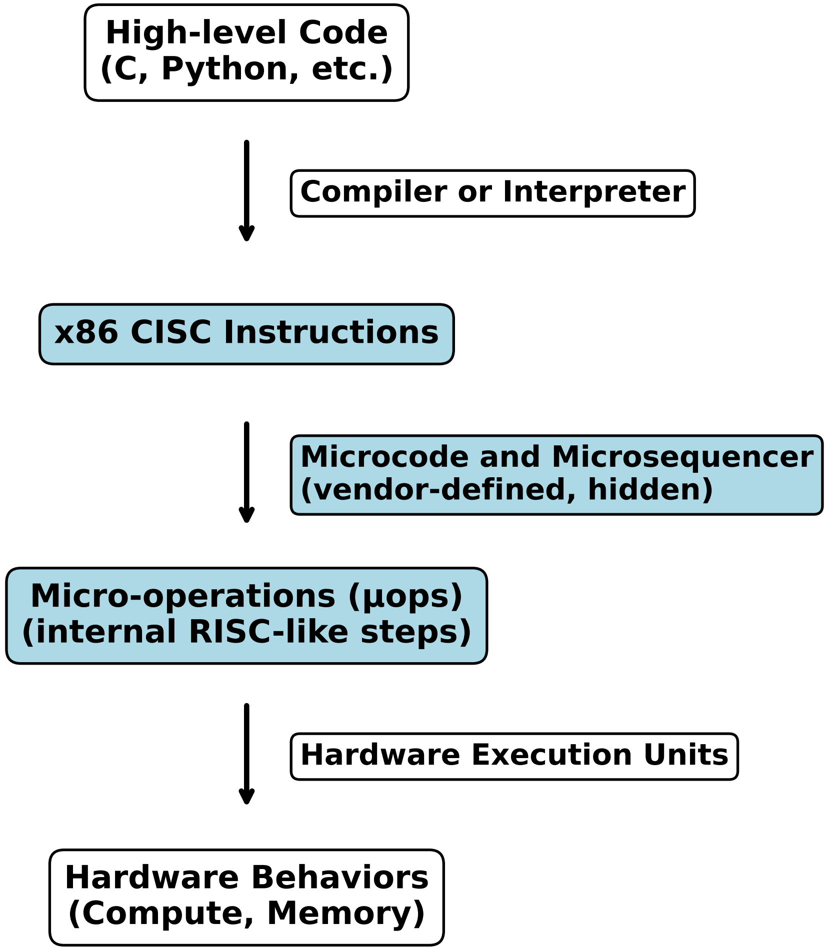
- Function: Translates **ISA instructions** into sequences of **micro-operations (µops)**. Many complex instructions are implemented this way.



What is Microcode?

- Function: Translates **ISA instructions** into sequences of **micro-operations (μops)**. Many complex instructions are implemented this way.

- Role: Works like firmware for the processor. **Defines** how each **instruction behaves**.



Why OS Researchers Should Care

Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.

Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.
- If microcode can be modified:

Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.

- If microcode can be modified:
  - **Optimize performance;**

Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.

- If microcode can be modified:
  - **Optimize performance;**
  - Explore future **ISA extensions;**

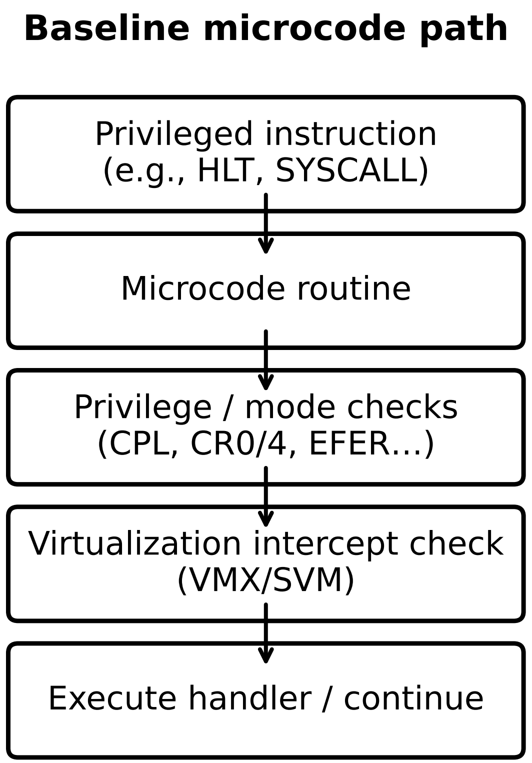
Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.

- If microcode can be modified:
  - **Optimize performance;**
  - Explore future **ISA extensions;**
  - Study security and isolation boundaries.

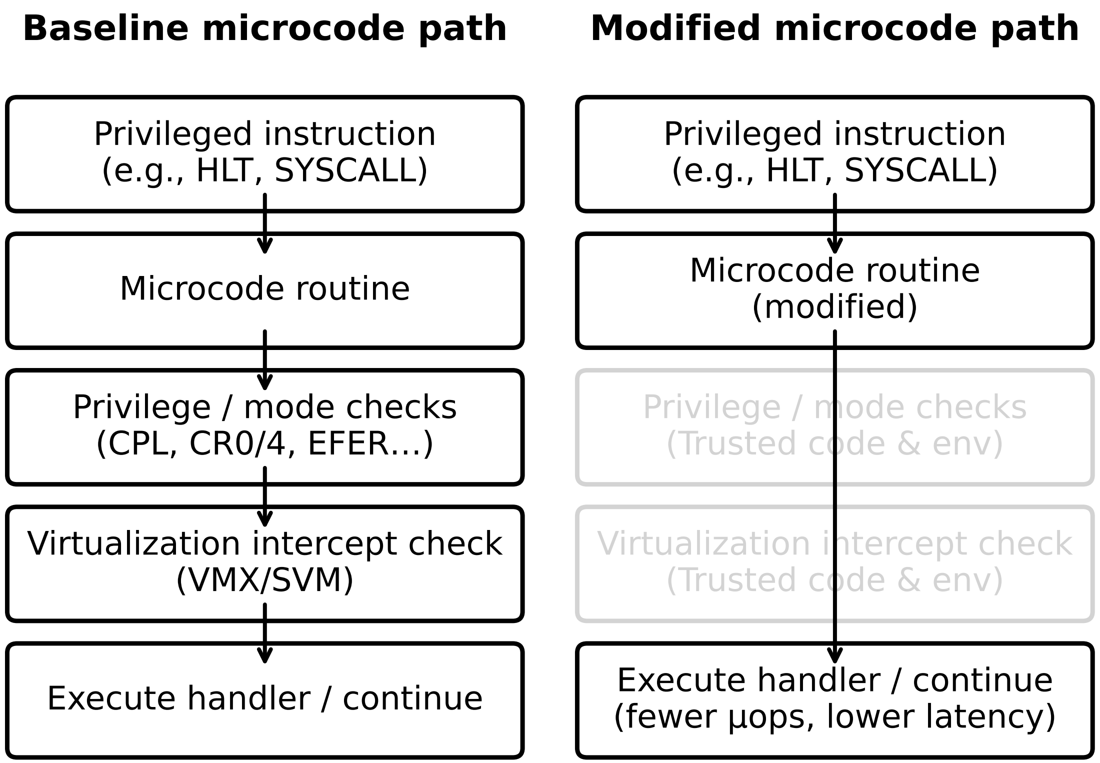
Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.
- If microcode can be modified:
  - **Optimize performance;**
  - Explore future **ISA extensions;**
  - Study security and isolation boundaries.



Why OS Researchers Should Care

- OS kernels heavily depend on instruction semantics: SYSCALL, privilege checks.
- If microcode can be modified:
  - **Optimize performance;**
  - Explore future **ISA extensions;**
  - Study security and isolation boundaries.



The Status Quo: Locked by Vendors 🔒

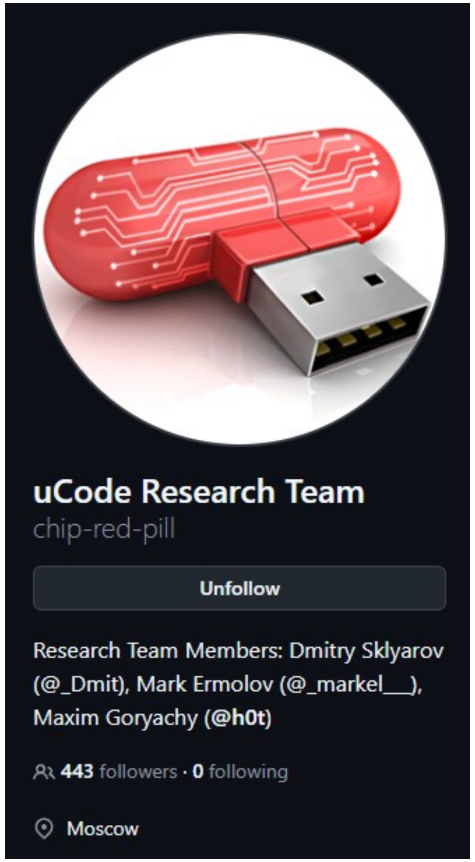
- Microcode is **proprietary**: no documentation, updates cryptographically signed, only distributed via vendor BIOS/OS channels.

- System software and researchers **cannot observe or modify** instruction behavior.

- So, the problem is, if we want to observe or modify the microcode layer, it is very difficult.

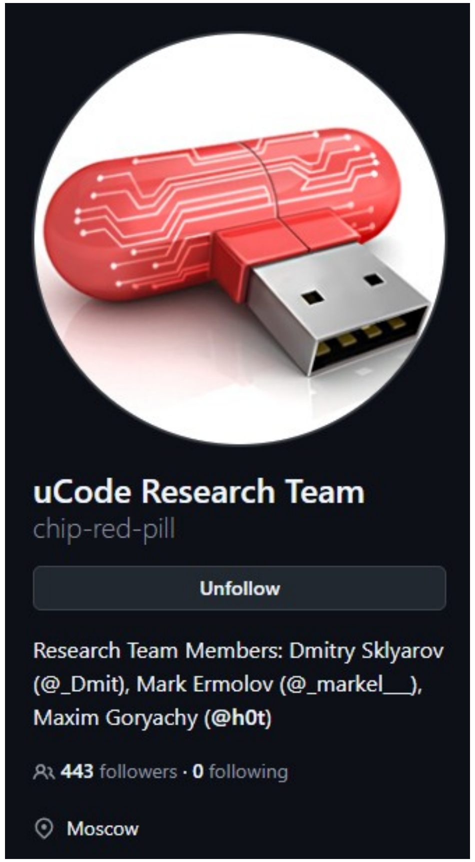
Emerging Possibility

Emerging Possibility



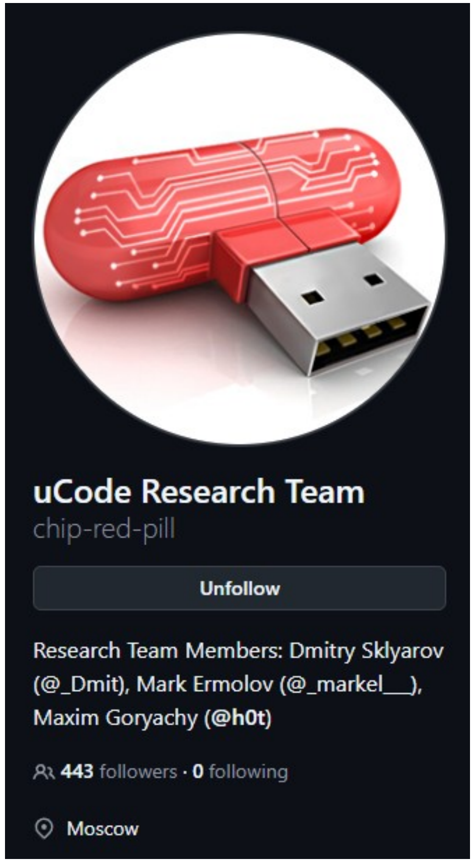
Emerging Possibility

- **INTEL-SA-00086:** grants privilege escalation.



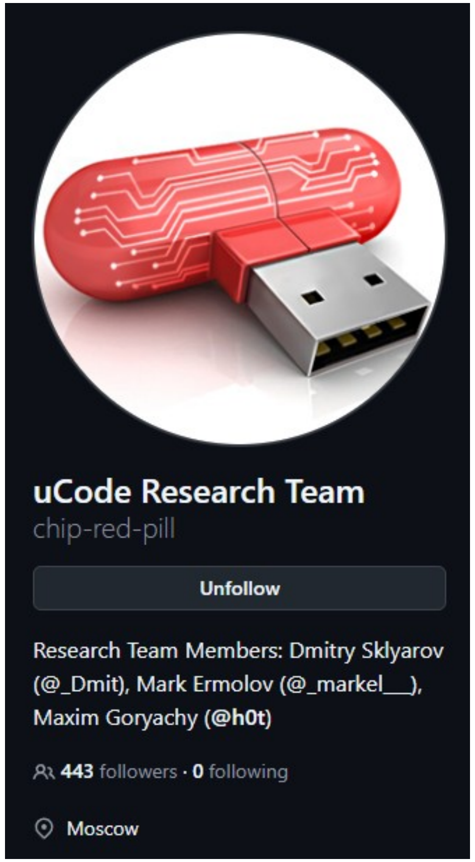
Emerging Possibility

- **INTEL-SA-00086:** grants privilege escalation.
- **CHIP-RED-PILL** enabled a hidden "**Red Unlock**" mode to access CPU microcode internals.



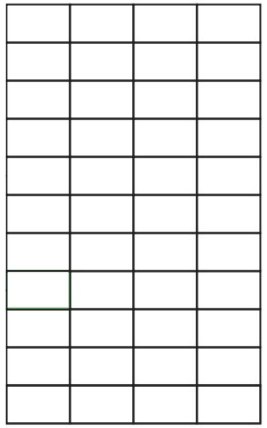
Emerging Possibility

- **INTEL-SA-00086:** grants privilege escalation.
- **CHIP-RED-PILL** enabled a hidden "**Red Unlock**" mode to access CPU microcode internals.
- **CHIP-RED-PILL** revealed the vendor µop format and encodings, which made authoring custom microcode feasible.



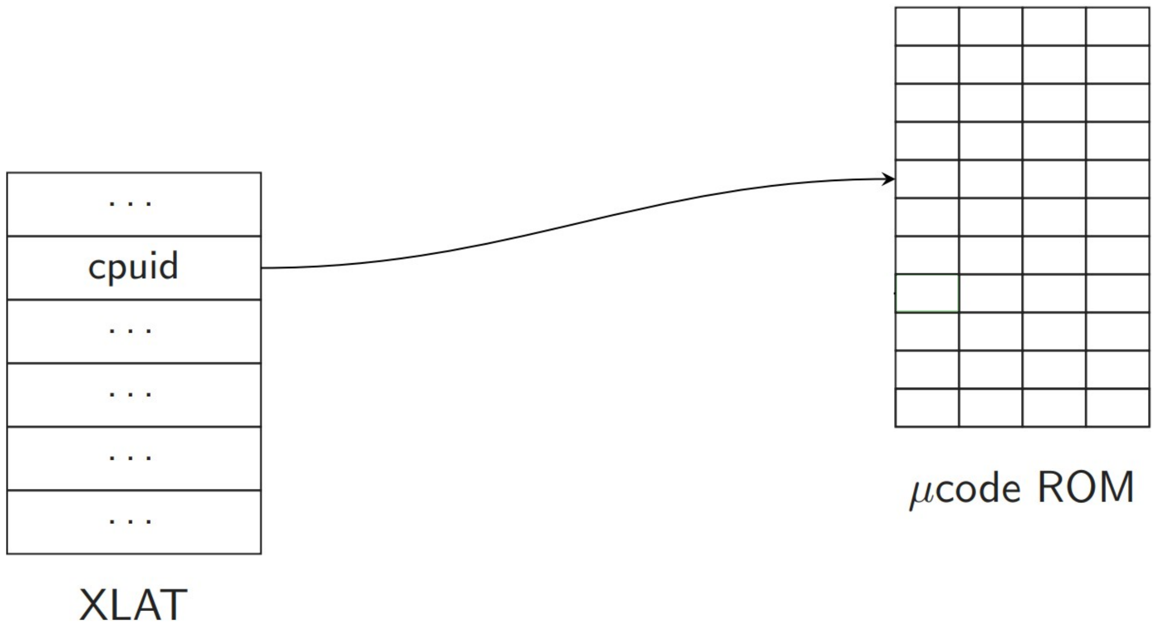
Understanding Goldmont’s Microcode Mechanism

Understanding Goldmont’s Microcode Mechanism

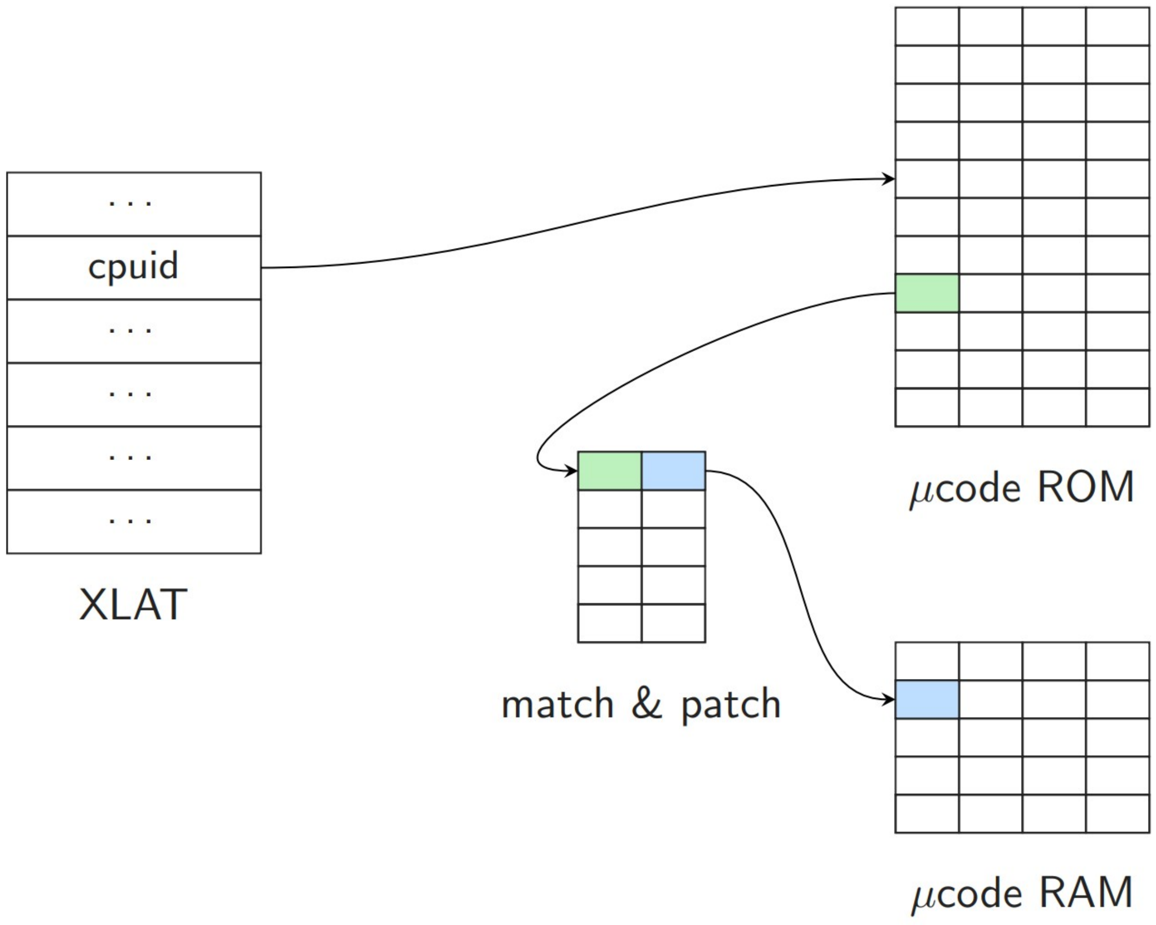


μcode ROM

Understanding Goldmont’s Microcode Mechanism



Understanding Goldmont’s Microcode Mechanism



Prior Microcode Research Efforts

Work	Main Contributions
CHIP-RED-PILL (Ermolov et al., 2021)	Red Unlock, µop Structure

Prior Microcode Research Efforts

Work	Main Contributions
CHIP-RED-PILL (Ermolov et al., 2021)	Red Unlock, µop Structure
CustomProcessingUnit (Borrello et al., WOOT 23)	UEFI-based patch and trace framework

Prior Microcode Research Efforts

Work	Main Contributions	Limitation
CHIP-RED-PILL (Ermolov et al., 2021)	Red Unlock, µop Structure	Unlock Only
CustomProcessingUnit (Borrello et al., WOOT 23)	UEFI-based patch and trace framework	

Prior Microcode Research Efforts

Work	Main Contributions	Limitation
CHIP-RED-PILL (Ermolov et al., 2021)	Red Unlock, µop Structure	Unlock Only
CustomProcessingUnit (Borrello et al., WOOT 23)	UEFI-based patch and trace framework	**Only runs in UEFI**; patches disappear after boot; no SMP support.

Prior Microcode Research Efforts

Work	Main Contributions	Limitation
CHIP-RED-PILL (Ermolov et al., 2021)	Red Unlock, μop Structure	Unlock Only
CustomProcessingUnit (Borrello et al., WOOT 23)	UEFI-based patch and trace framework	**Only runs in UEFI**; patches disappear after boot; no SMP support.

These efforts proved that **microcode can be modified.**
But **not in a standard Linux runtime environment.**

Motivation for Our Work

Motivation for Our Work

- We need a **practical, OS-integrated framework** that brings **custom microcode** into normal **Linux** systems.

Motivation for Our Work

- We need a **practical, OS-integrated framework** that brings **custom microcode** into normal **Linux** systems.

- Need to support:
  - **Multi-core (SMP)** patching;

Motivation for Our Work

- We need a **practical, OS-integrated framework** that brings **custom microcode** into normal **Linux** systems.

- Need to support:
  - **Multi-core (SMP)** patching;
  - **Runtime updates** without reboot;
  - **Fine-grained** microcode management;

Motivation for Our Work

- We need a **practical, OS-integrated framework** that brings **custom microcode** into normal **Linux** systems.

- Need to support:
  - **Multi-core (SMP)** patching;
  - **Runtime updates** without reboot;
  - **Fine-grained** microcode management;
  - **User-space interface.**

Where we built on prior work

Where we built on prior work

- We adopted **CHIP-RED-PILL**’s idea to **unlock** our board’s hidden debug interfaces (enables MSRAM / match table access).

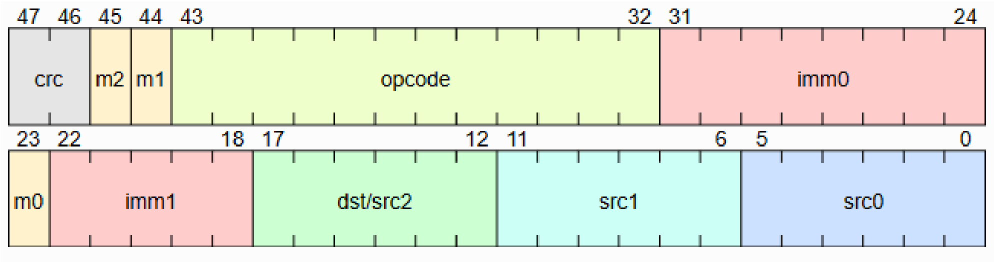
Where we built on prior work

- We adopted **CHIP-RED-PILL**'s idea to **unlock** our board's hidden debug interfaces (enables MSRAM / match table access).
- We adopted **CustomProcessingUnit**'s microcode toolchain (DSL and assembler/serializer).

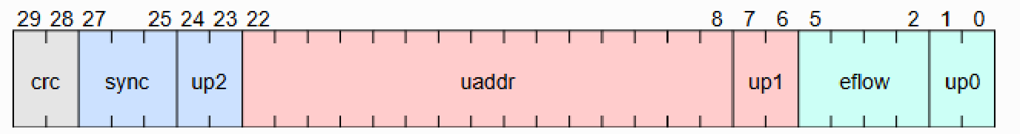
```
labuser@bmax-1:~/microcode-project/ucode_patches$ cat rdrand.u
.org 0x7c00
.patch 0x428
.entry 0

r64src := ZEROEXT_DSZ64(0x1146)
labuser@bmax-1:~/microcode-project/ucode_patches$ cat rdrand.h
struct ucode_patch_req rdrand_req = {
    .name = "rdrand",
    .addr = 0x7c00,
    .hook_address = 0x0428,
    .hook_entry = 0x00,
    .patch_rows = 1,
    .patch_data = {
        // U7c00: r64src := ZEROEXT_DSZ64(0x1146); unk_256() !m1; NOP SEQW LFNCEWAIT, UEND0
        {0x404846442008, 0x125600000000, 0x0, 0x130000f2},
    }
};
```

µop:



Seqword:



Our **Framework** in **Linux**

Our **Framework** in **Linux**

- Disabled Linux’s built-in microcode subsystem:

Our **Framework** in **Linux**

- Disabled Linux’s built-in microcode subsystem:
  - Removed boot-time microcode update.

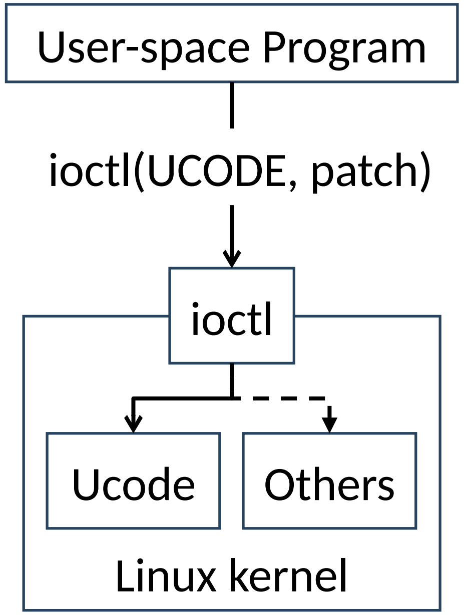
Our **Framework** in **Linux**

- Disabled Linux’s built-in microcode subsystem:
  - Removed boot-time microcode update.
  - Disabled CPU C-states to retain patches.

Our **Framework** in **Linux**

- Disabled Linux’s built-in microcode subsystem:
  - Removed boot-time microcode update.
  - Disabled CPU C-states to retain patches.

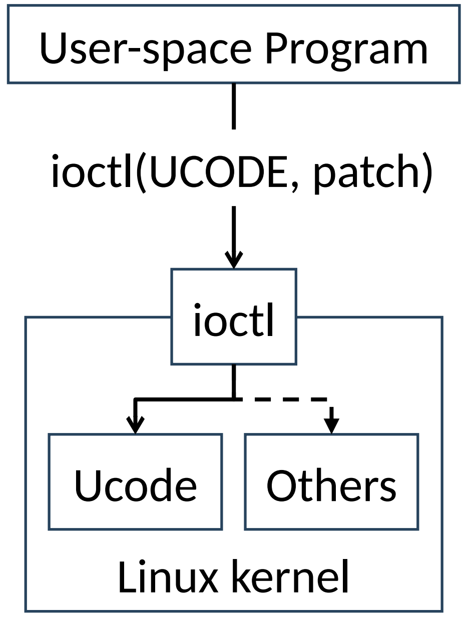
- Developed a Linux kernel module “**Ucode**”:



Our **Framework** in **Linux**

- Disabled Linux's built-in microcode subsystem:
  - Removed boot-time microcode update.
  - Disabled CPU C-states to retain patches.

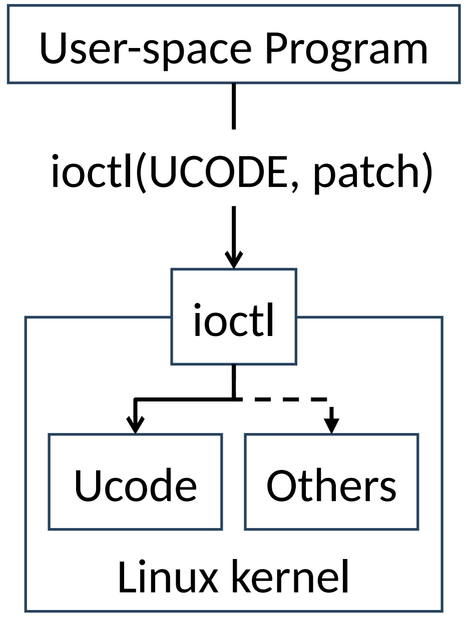
- Developed a Linux kernel module "**Ucode**":
  - Exposes a user-space ioctl interface;



Our **Framework** in **Linux**

- Disabled Linux's built-in microcode subsystem:
  - Removed boot-time microcode update.
  - Disabled CPU C-states to retain patches.

- Developed a Linux kernel module "**Ucode**":
  - Exposes a user-space ioctl interface;
  - Accepts and applies custom microcode patches.



Microcode-Based Kernel Memory Access

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

- We replaced the microcode of the RDRAND instruction:
  - Original behavior: return a random number.

```c
#define rd(addr) ({ \
    unsigned long long __res; \
    asm volatile( \
        "rdrand %%rbx\n" \
        : "=b"(__res) \
        : "a"(addr) \
    ); \
    __res; \
})
```

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

- We replaced the microcode of the RDRAND instruction:
  - Original behavior: return a random number.
  - Modified behavior: RBX := *(uint64_t *)RAX

```
#define rd(addr) ({ \
    unsigned long long __res; \
    asm volatile( \
        "rdrand %%rbx\n" \
        : "=b"(__res) \
        : "a"(addr) \
    ); \
    __res; \
})
```

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

- We replaced the microcode of the RDRAND instruction:
  - Original behavior: return a random number.
  - Modified behavior: RBX := *(uint64_t *)RAX

- Key Properties:
  - No privilege checks: bypasses CPL, SMEP, SMAP;

```
#define rd(addr) ({ \
    unsigned long long __res; \
    asm volatile( \
        "rdrand %%rbx\n" \
        : "=b"(__res) \
        : "a"(addr) \
    ); \
    __res; \
})
```

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

- We replaced the microcode of the RDRAND instruction:
  - Original behavior: return a random number.
  - Modified behavior: RBX := *(uint64_t *)RAX

- Key Properties:
  - No privilege checks: bypasses CPL, SMEP, SMAP;
  - Works on kernel addresses (0xffff…);

```
#define rd(addr) ({ \
    unsigned long long __res; \
    asm volatile( \
        "rdrand %%rbx\n" \
        : "=b"(__res) \
        : "a"(addr) \
    ); \
    __res; \
})
```

Microcode-Based Kernel Memory Access

- Goal: Enable a **user-space instruction** to read from **any virtual address**, including kernel space, without triggering faults or access checks.

- We replaced the microcode of the RDRAND instruction:
  - Original behavior: return a random number.
  - Modified behavior: RBX := *(uint64_t *)RAX

- Key Properties:
  - No privilege checks: bypasses CPL, SMEP, SMAP;
  - Works on kernel addresses (0xffff…);
  - Silent execution: does not raise exceptions or faults.

```
#define rd(addr) ({ \
    unsigned long long __res; \
    asm volatile( \
        "rdrand %%rbx\n" \
        : "=b"(__res) \
        : "a"(addr) \
    ); \
    __res; \
})
```

Preliminary Results: Reading Process List

Preliminary Results: Reading Process List

- Objective:
  - Enumerate all processes in the system by traversing the kernel’s task_struct list.

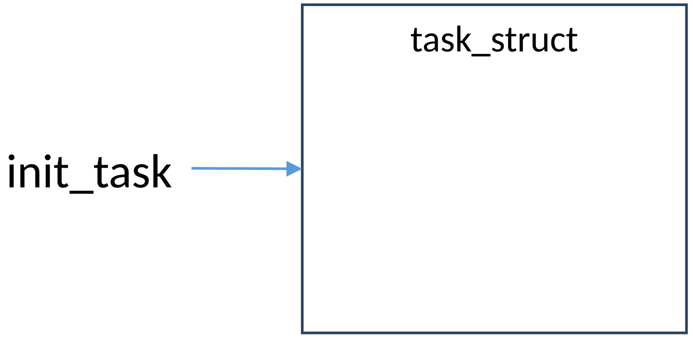
Preliminary Results: Reading Process List

- Objective:
  - Enumerate all processes in the system by traversing the kernel's task_struct list.
  - Demonstrate ability to extract arbitrary fields from each task.

Preliminary Results: Reading Process List

- Objective:
  - Enumerate all processes in the system by traversing the kernel's task_struct list.
  - Demonstrate ability to extract arbitrary fields from each task.

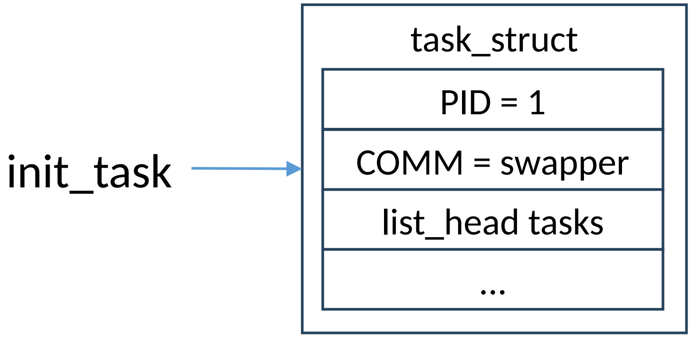
- Method Overview (1st part):
  - Locate the symbol init_task, which gives us the address of the first task_struct.



Preliminary Results: Reading Process List

- Objective:
  - Enumerate all processes in the system by traversing the kernel's task_struct list.
  - Demonstrate ability to extract arbitrary fields from each task.

- Method Overview (1st part):
  - Locate the symbol init_task, which gives us the address of the first task_struct.
  - Once a task_struct is found, we can **extract fields by their known offsets:**
    - For example, to read the process name: comm = *(task_addr + comm_offset)

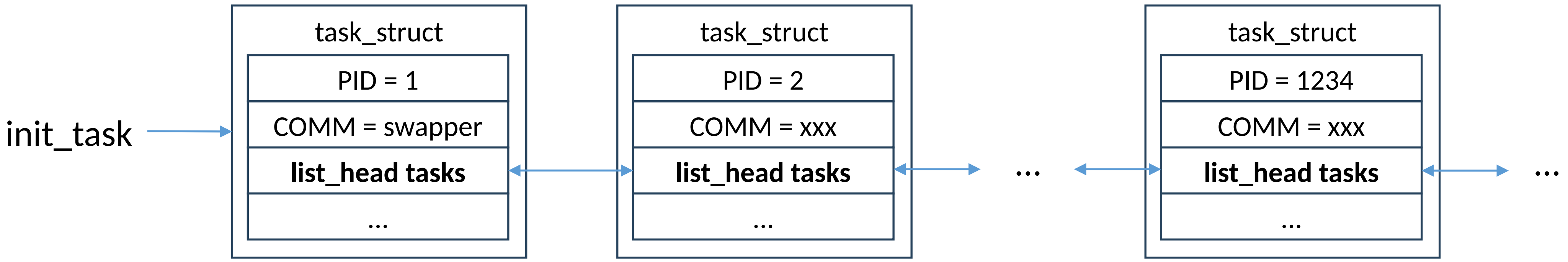


Preliminary Results: Reading Process List

- Method Overview (2nd part):

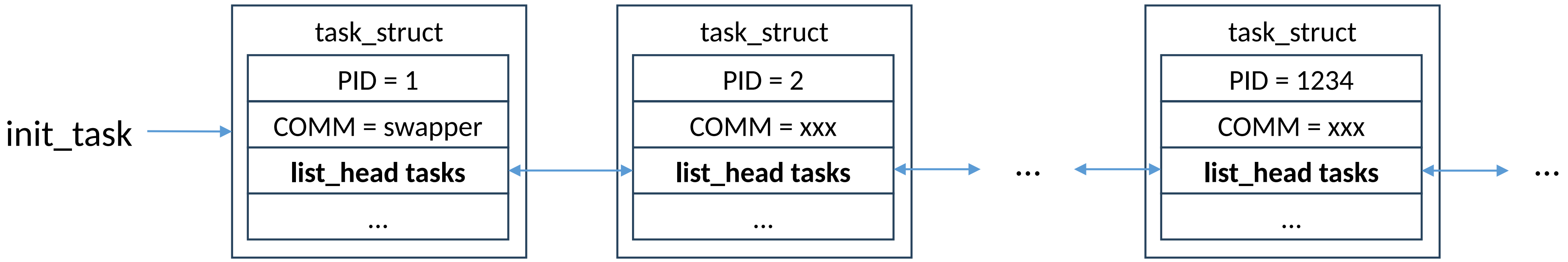
Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.



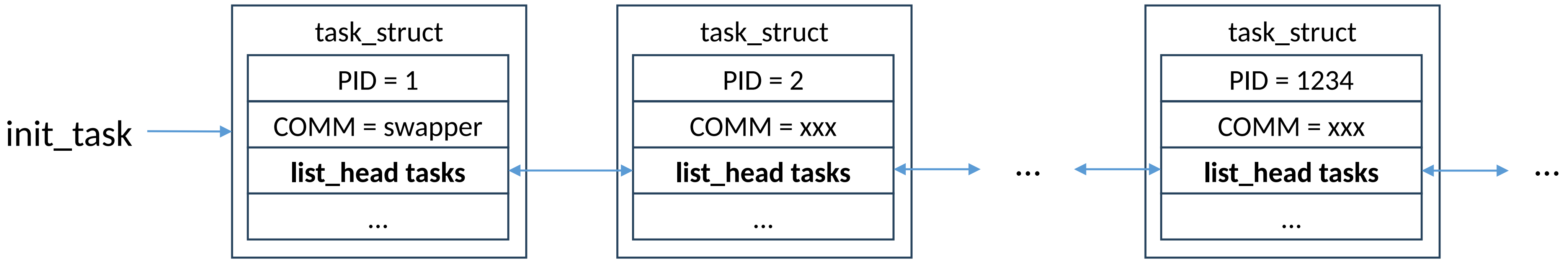
Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.



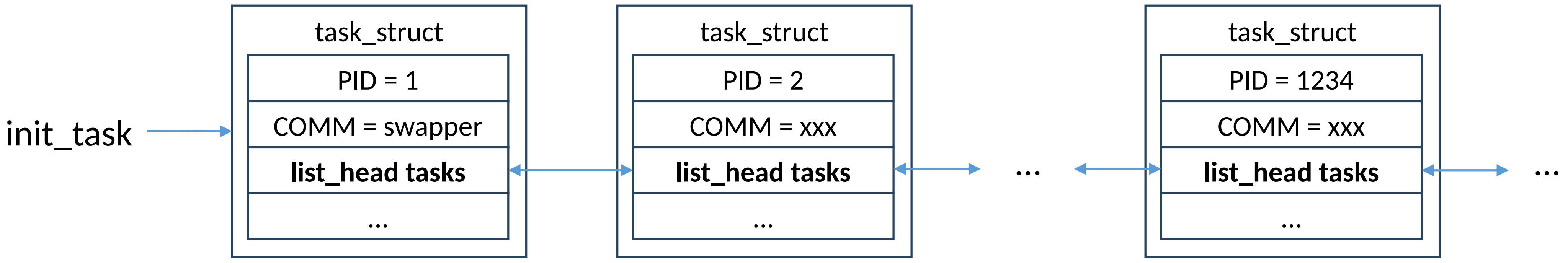
Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.
  - By reading **tasks.next** (at a known offset), we can:



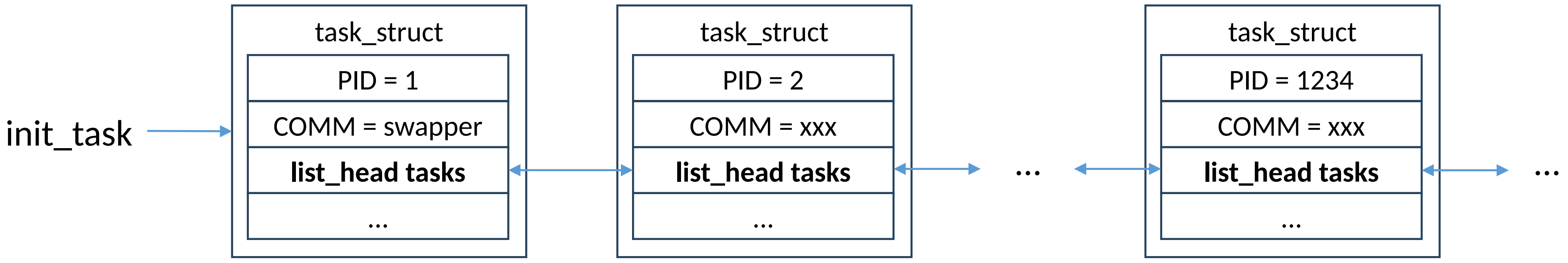
Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.
  - By reading **tasks.next** (at a known offset), we can:
    - Walk to the next task;
    - Repeat until we loop back to init_task;



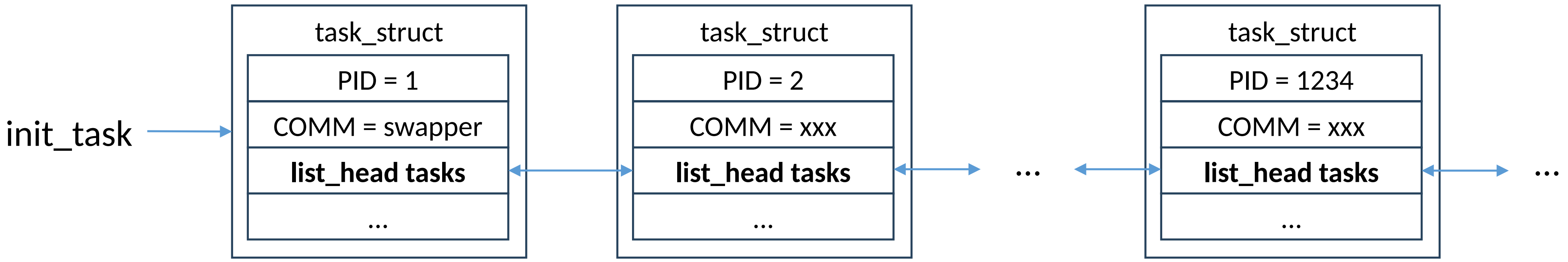
Preliminary Results: Reading Process List

- Method Overview (2^{nd} part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.
  - By reading **tasks.next** (at a known offset), we can:
    - Walk to the next task;
    - Repeat until we loop back to init_task;
    - Iterate through all tasks and print their comm (process name).



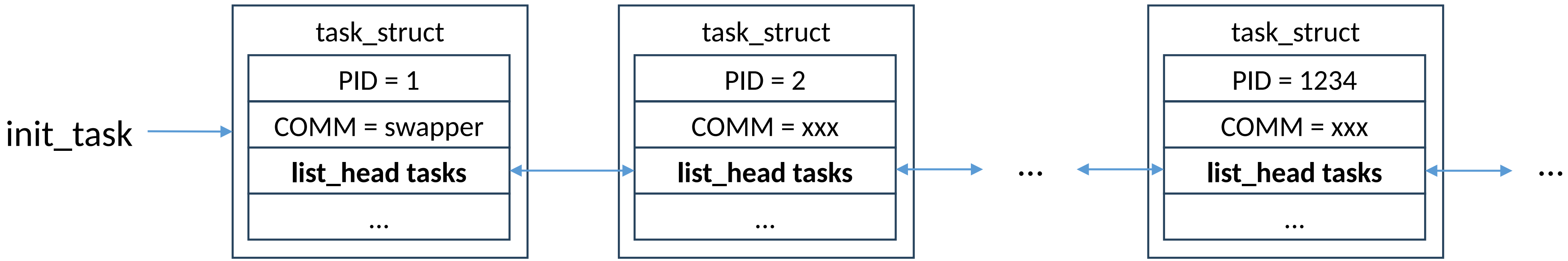
Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.
  - By reading **tasks.next** (at a known offset), we can:
    - Walk to the next task;
    - Repeat until we loop back to init_task;
    - Iterate through all tasks and print their comm (process name).



Preliminary Results: Reading Process List

- Method Overview (2nd part):
  - Linux links all tasks via a **circular doubly-linked list** using **task_struct.tasks**.
  - By reading **tasks.next** (at a known offset), we can:
    - Walk to the next task;
    - Repeat until we loop back to init_task;
    - Iterate through all tasks and print their comm (process name).



Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11  COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12  COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13  COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14  COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15  COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16  COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17  COMM = rcu_preempt
```

Preliminary Results: Reading Process List

```
labuser@bmax-1:~/microcode-project$ make test_read_tasks
gcc test_read_tasks.c -o test_read_tasks.o
./test_read_tasks.o
Patch applied successfully!
== init_task ==
Task @ 0xffffffffab010940: PID = 0, COMM = swapper/0

== Iterating through all task_struct ==
Task @ 0xffff974e402a3200: PID = 1, COMM = systemd
Task @ 0xffff974e402a0000: PID = 2, COMM = kthreadd
Task @ 0xffff974e402a6400: PID = 3, COMM = pool_workqueue_
Task @ 0xffff974e402a1900: PID = 4, COMM = kworker/R-rcu_g
Task @ 0xffff974e402a4b00: PID = 5, COMM = kworker/R-sync_
Task @ 0xffff974e4032b200: PID = 6, COMM = kworker/R-slub_
Task @ 0xffff974e40328000: PID = 7, COMM = kworker/R-netns
Task @ 0xffff974e40330000: PID = 11, COMM = kworker/u8:0
Task @ 0xffff974e40336400: PID = 12, COMM = kworker/R-mm_pe
Task @ 0xffff974e40331900: PID = 13, COMM = rcu_tasks_kthre
Task @ 0xffff974e40334b00: PID = 14, COMM = rcu_tasks_rude_
Task @ 0xffff974e40333200: PID = 15, COMM = rcu_tasks_trace
Task @ 0xffff974e4033e400: PID = 16, COMM = ksoftirqd/0
Task @ 0xffff974e40339900: PID = 17, COMM = rcu_preempt
```

Preliminary Results: Reading **sudo Password**

Preliminary Results: Reading **sudo Password**

- Objective:
  - From a **user-space program**, extract the **password** typed into a **sudo** prompt;

Preliminary Results: Reading **sudo Password**

- Objective:
  - From a **user-space program**, extract the **password** typed into a **sudo** prompt;
  - Target the **input buffer** of the **tty** associated with the **sudo** process.

Preliminary Results: Reading **sudo Password**

- Objective:
  - From a **user-space program**, extract the **password** typed into a **sudo** prompt;
  - Target the **input buffer** of the **tty** associated with the **sudo** process.

- Method Overview:
  - Use previously described technique to find the sudo process;

Preliminary Results: Reading **sudo Password**

- Objective:
  - From a **user-space program**, extract the **password** typed into a **sudo** prompt;
  - Target the **input buffer** of the **tty** associated with the **sudo** process.

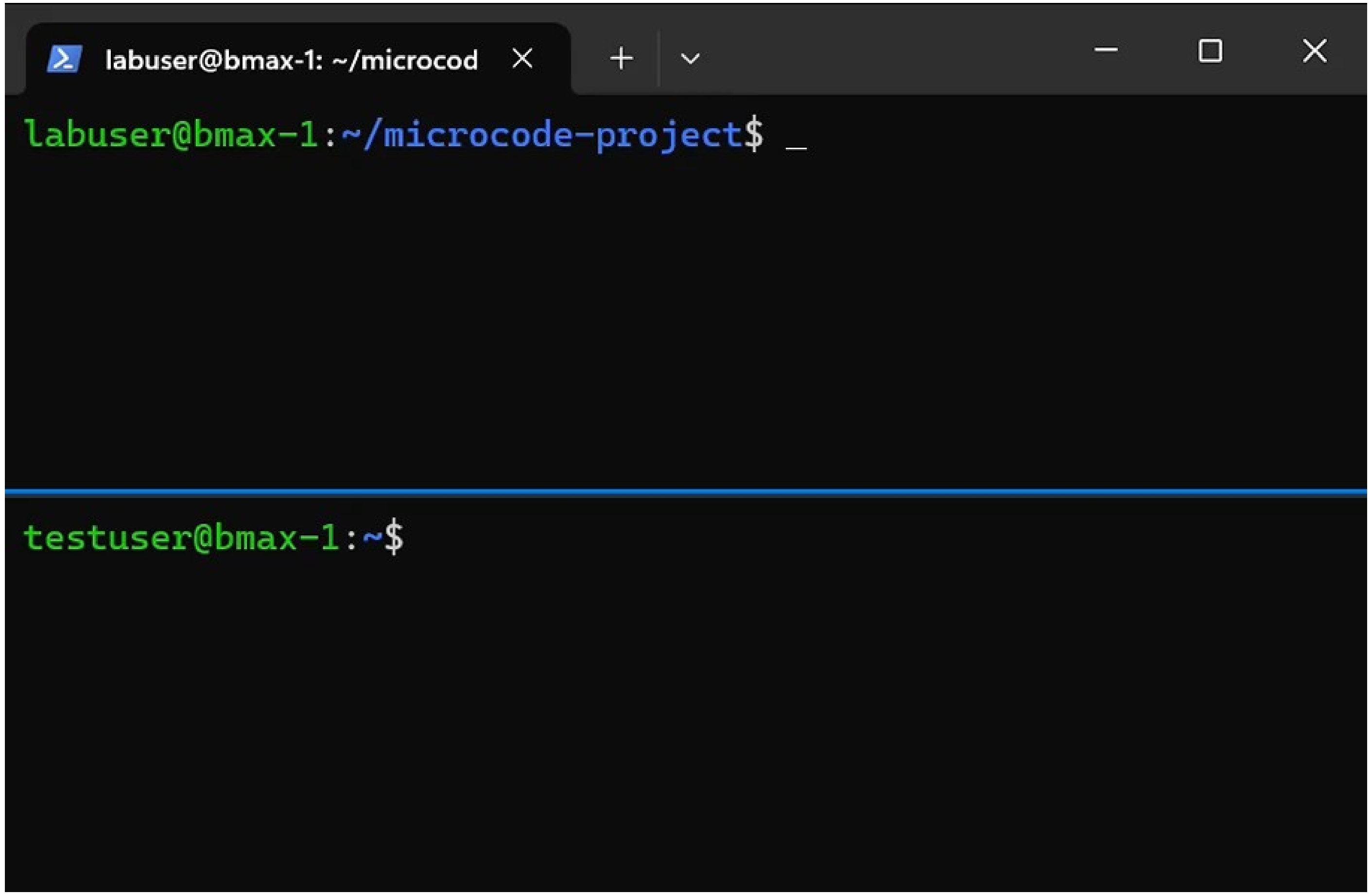
- Method Overview:
  - Use previously described technique to find the sudo process;
  - Navigate to its tty input buffer via the following pointer chain:
    - task_struct → files_struct → fdtable → file[0] → private_data → tty_struct → ldisc → n_tty_data → read_buf

Preliminary Results: Reading sudo Password

- Objective:
  - From a **user-space program**, extract the **password** typed into a **sudo** prompt;
  - Target the **input buffer** of the **tty** associated with the **sudo** process.

- Method Overview:
  - Use previously described technique to find the sudo process;
  - Navigate to its tty input buffer via the following pointer chain:
    - task_struct → files_struct → fdtable → file[0] → private_data → tty_struct → ldisc → n_tty_data → read_buf
  - Read the password from the buffer.

Preliminary Results: Reading **sudo Password**



Preliminary Results: getcpu() Syscall Acceleration

Preliminary Results: getcpu() Syscall Acceleration

- A typical syscall involves: user → trap → kernel entry → kernel work → return → user.

Preliminary Results: getcpu() Syscall Acceleration

- A typical syscall involves: user → trap → kernel entry → kernel work → return → user.

- Mode switches + stack/frame setup + validation = **hundreds of cycles** for small queries.

Preliminary Results: getcpu() Syscall Acceleration

- A typical syscall involves: user → trap → kernel entry → kernel work → return → user.
- Mode switches + stack/frame setup + validation = **hundreds of cycles** for small queries.
- Many OS queries are tiny (e.g., getcpu(), getpid(), getuid()), yet pay full syscall cost.

Preliminary Results: getcpu() Syscall Acceleration

- A typical syscall involves: user → trap → kernel entry → kernel work → return → user.
- Mode switches + stack/frame setup + validation = **hundreds of cycles** for small queries.
- Many OS queries are tiny (e.g., getcpu(), getpid(), getuid()), yet pay full syscall cost.
- Example: getcpu() syscall measured ≈ **511 cycles** (baseline in our experiments).

Preliminary Results: getcpu() Syscall Acceleration

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.
- Per-core patches: install different constant-return microcode on each logical CPU.

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.
- Per-core patches: install different constant-return microcode on each logical CPU.
  - On CPU0: rax=ZEROEXT(0)

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.
- Per-core patches: install different constant-return microcode on each logical CPU.
  - On CPU0: rax=ZEROEXT(0)
  - On CPU1: rax=ZEROEXT(1)

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.
- Per-core patches: install different constant-return microcode on each logical CPU.
  - On CPU0: rax=ZEROEXT(0)
  - On CPU1: rax=ZEROEXT(1)
  - On CPU2: rax=ZEROEXT(2) …

Preliminary Results: getcpu() Syscall Acceleration

- Idea: replace a user-visible instruction (we use RDRAND) with a per-core microcode patch that returns the core ID.
- Per-core patches: install different constant-return microcode on each logical CPU.
  - On CPU0: rax=ZEROEXT(0)
  - On CPU1: rax=ZEROEXT(1)
  - On CPU2: rax=ZEROEXT(2) …
- User program executes RDRAND → immediate core ID returned (no syscall, no kernel transition).

Preliminary Results: getcpu() Syscall Acceleration

- Microcode version eliminates syscall transition overhead.

Preliminary Results: getcpu() Syscall Acceleration

- Microcode version eliminates syscall transition overhead.

Method	Description	Latency (cycles)	Speed-up
`getcpu()` system call	User → kernel → scheduler lookup → return	**511**	1×

Preliminary Results: getcpu() Syscall Acceleration

- Microcode version eliminates syscall transition overhead.

Method	Description	Latency (cycles)	Speed-up
`getcpu()` system call	User → kernel → scheduler lookup → return	**511**	1×
Microcode version	Direct constant return from patched `RDRAND`	**43**	**≈12×**

Preliminary Results: getcpu() Syscall Acceleration

- Microcode version eliminates syscall transition overhead.

Method	Description	Latency (cycles)	Speed-up
`getcpu()` system call	User → kernel → scheduler lookup → return	**511**	1×
Microcode version	Direct constant return from patched `RDRAND`	**43**	**≈12×**
Native `RDRAND`	Hardware random generator baseline	**9**	N/A

Preliminary Results: getcpu() Syscall Acceleration

- Microcode version eliminates syscall transition overhead.

Method	Description	Latency (cycles)	Speed-up
`getcpu()` system call	User → kernel → scheduler lookup → return	**511**	1×
Microcode version	Direct constant return from patched `RDRAND`	**43**	**≈12×**
Native `RDRAND`	Hardware random generator baseline	**9**	N/A

- Demonstrates microcode as a viable fast path for frequent syscalls.

Future Work

Future Work

- **More Available μops**
  Extend patching to more instruction types, enabling richer functionality and finer performance control.

Future Work

- **More Available µops**
  Extend patching to more instruction types, enabling richer functionality and finer performance control.
- **Better Framework**
  Build a microcode framework that supports **per-core** and **per-thread** customization, integrates with the Linux kernel, and provides easier user-space access.

Future Work

- **More Available μops**
  Extend patching to more instruction types, enabling richer functionality and finer performance control.
- **Better Framework**
  Build a microcode framework that supports **per-core** and **per-thread** customization, integrates with the Linux kernel, and provides easier user-space access.
- **AMD Support**
  Port and test on AMD Zen CPUs to explore cross-vendor compatibility and new customization opportunities.

Q & A

Thank you!

This work is supported in part by the US Office of Naval Research (ONR) under grant N000142412642.